(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 March 2003 (20.03.2003)

PCT

(10) International Publication Number
WO 03/023647 A1

(51) International Patent Classification[7]: G06F 17/00

(21) International Application Number: PCT/US02/30286

(22) International Filing Date:
10 September 2002 (10.09.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/318,369 10 September 2001 (10.09.2001) US

(71) Applicant: **SHUFFLE MASTER, INC.** [US/US]; 1106 Palms Airport Drive, Las Vegas, NV 89119 (US).

(72) Inventor: **JACKSON, Mark, D.**; 3902 Ridgeway Court, Fort Collins, CO 80526 (US).

(74) Agent: **FARRAR, Jennifer, K.**; Shuffle Master, Inc., 1106 Palms Airport Drive, Las Vegas, NV 89119 (US).

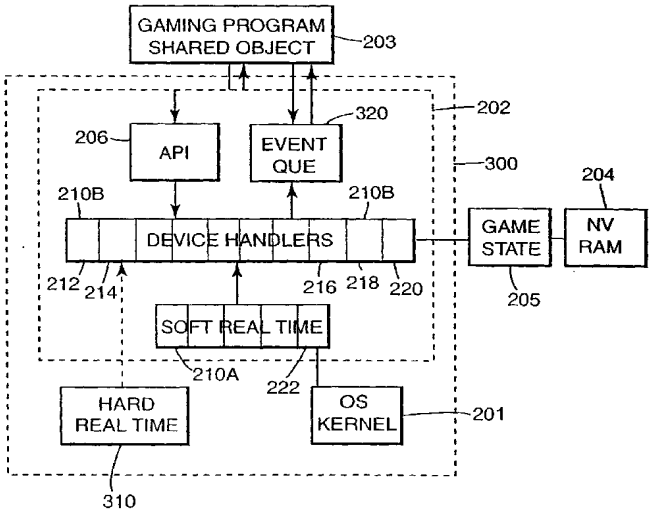
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD FOR DEVELOPING GAMING PROGRAMS COMPATIBLE WITH A COMPUTERIZED GAMING OPERATING SYSTEM AND APPARATUS



(57) Abstract: The present invention in various embodiments provides a computerized wagering game method and apparatus that features an operating system kernel (201), a system handler application (202) that loads and executes gaming program shared objects (203) and features nonvolatile storage (204) that facilitates sharing of information between gaming program objects (203). The system handler of some embodiments further provides an API library of functions callable from the gaming program objects (203), and facilitates the use of callback functions on change of data stored in nonvolatile storage (204). The nonvolatile storage (204) also provides a nonvolatile record of the state of the computerized wagering game, providing protection against loss of the game state due to power loss. The system handler application (202) in various embodiments includes a plurality of device handlers (210), providing an interface to selected hardware and the ability to monitor hardware-related events.

WO 03/023647 A1

METHOD FOR DEVELOPING GAMING PROGRAMS COMPATIBLE WITH A COMPUTERIZED GAMING OPERATING SYSTEM AND APPARATUS

NOTICE OF COPENDING APPLICATIONS

This application is a non-provisional application claiming priority from Provisional Patent Application Serial No. 60/318,369 filed September 10, 2001, entitled: Method for Developing Gaming Programs Compatible with a Computerized Gaming Operating System and Apparatus (Attorney Docket No. PA0616.ap.US).

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to wagering games, particularly computer based wagering games, computer based wagering games running on an operating system, and methods for developing games on a standard gaming operating system.

2. Background of the Art

Games of chance have been enjoyed by people for thousands of years and have enjoyed increased and widespread popularity in recent times. As with most forms of entertainment, players enjoy playing a wide variety of games and new games. Playing new games adds to the excitement of "gaming." As is well known in the art and as used herein, the term "gaming" and "gaming devices" are used to indicate that some form of wagering is involved, and that players must make wagers of value, whether actual currency or some equivalent of value, e.g., token or credit. This is an accepted distinction in the art from the playing of games, which implies the lack of value depending upon the outcome and in which skill is ordinarily an essential part of the game. On the contrary, within the gaming industry, particularly in computer based gaming systems, the absence of skill is a jurisdictional requirement in the performance of the gaming play.

One popular gaming system of chance is the slot machine. Conventionally, a slot machine is configured for a player to wager something of value, e.g., currency, house token, established credit or other representation of currency or credit. After the

wager has been made, the player activates the slot machine to cause a random event to occur. The player wagers that particular random events will occur that will return value to the player. A standard device causes a plurality of reels to spin and ultimately stop, displaying a random combination of some form of indicia, for example, numbers or symbols. If this display contains one of a pre-selected number of winning symbol combinations, the machine releases money into a payout chute or increments a credit meter by the amount won by the player. For example, if a player initially wagered two coins of a specific denomination and that player achieved a payout, that player may receive the same number or multiples of the wager amount in coins of the same denomination as wagered.

There are many different formats for generating the random display of events that can occur to determine payouts in wagering devices. The standard or original format was the use of three reels with symbols distributed over the face of the reel. When the three reels were spun, they would eventually each stop in turn, displaying a combination of three symbols (e.g., with three reels and the use of a single horizontal payout line as a row in the middle of the area where the symbols are displayed). By appropriately distributing and varying the symbols on each of the reels, the random occurrence of predetermined winning combinations can be provided in mathematically predetermined probabilities. By clearly providing for specific probabilities for each of the pre-selected winning outcomes, precise odds that would control the amount of the payout for any particular combination and the percentage return on wagers for the house could be reasonably controlled.

Other formats of gaming apparatus that have developed in a progression from the pure slot machine with three reels have dramatically increased with the development of video gaming apparatus. Rather than have only mechanical elements such as wheels or reels that turn and stop to randomly display symbols, video gaming apparatus and the rapidly increasing sophistication in hardware and software have enabled an explosion of new and exciting gaming apparatus. The earlier video apparatus merely imitated or simulated the mechanical slot games in the belief that players would want to play only the same games. Early video gaming systems therefore were simulated slot machines. The use of video gaming apparatus to play

new gaming applications such as draw poker and Keno broke the ground for the realization that there were many untapped formats for gaming apparatus. Now casinos may have hundreds of different types of gaming apparatus with an equal number of significant differences in play. The apparatus may vary from traditional three reel slot machines with a single payout line, video simulations of three reel video slot machines, to five reel, five column simulated slot machines with a choice of twenty or more distinct pay lines, including randomly placed lines, scatter pays, or single image payouts. In addition to the variation in formats for the play of gaming applications, bonus plays, bonus awards, and progressive jackpots have been introduced with great success. The bonuses may be associated with the play of gaming applications that are quite distinct from the play of the original gaming format, such as the video display of a horse race with "bets" on the individual horses randomly assigned to players that qualify for a bonus, the spinning of a random wheel with fixed amounts of a bonus payout on the wheel (or simulation thereof), or attempting to select a random card that is of higher value than a card exposed on behalf of a virtual "dealer."

Examples of such gaming apparatus with a distinct bonus feature includes U.S. Patent Nos. 5,823,874; 5,848,932; 5,836,041; U.K. Patent Nos. 2 201 821 A; 2 202 984 A; and 2 072 395A; and German Patent DE 40 14 477 A1. Each of these patents differs in fairly subtle ways as to the manner in which the bonus round is played. British Patent 2 201 821 A and DE 37 00 861 A1 describe a gaming apparatus in which after a winning outcome is first achieved in a reel-type gaming segment, a second segment is engaged to determine the amount of money or extra games awarded. The second segment gaming play involves a spinning wheel with awards listed thereon (e.g., the number of coins or number of extra plays) and a spinning arrow that will point to segments of the wheel with the values of the awards thereon. A player will press a stop button and the arrow will point to one of the values. The specification indicates both that there is a level of skill possibly involved in the stopping of the wheel and the arrow(s), and also that an associated computer operates the random selection of the rotatable numbers and determines the results in

the additional winning game, which indicates some level of random selection in the second gaming segment.

U.S. Patents Nos. 5,823,874 and 5,848,932 describe a gaming device comprising:

a first, standard gaming unit for displaying a randomly selected combination of indicia, said displayed indicia selected from the group consisting of reels, indicia of reels, indicia of playing cards, and combinations thereof; means for generating at least one signal corresponding to at least one select display of indicia by said first, standard gaming unit; means for providing at least one discernible indicia of a mechanical bonus indicator, said discernible indicia indicating at least one of a plurality of possible bonuses, wherein said providing means is operatively connected to said first, standard gaming unit and becomes actuatable in response to said signal. In effect, the second gaming event simulates a mechanical bonus indicator such as a roulette wheel or wheel with a pointing element.

A video terminal is another form of gaming device. Video terminals operate in the same manner as a conventional slot and video machine, except that a redemption ticket rather than an immediate payout is dispensed. The processor may be present in the terminal or in a central computer.

The vast array of electronic video gaming apparatus that is commercially available is not standardized within the industry or necessarily even within the commercial line of apparatus available from a single manufacturer. One of the reasons for this lack of uniformity or standardization is the fact that the operating systems that have been used to date in the industry are primitive. As a result, the programmer must often create code for each and every function performed by each individual apparatus.

Attempts have been made to create a universal gaming engine for a gaming machine and are described in Carlson U.S. Patent 5,707,286. This patent describes a universal gaming engine that segregates the random number generator and transform algorithms so that this code need not be rewritten or retested with each new game application. All code that is used to generate a particular game is contained in a rule EPROM in the rules library. Although the step of segregating random number

generator code and transform algorithms has reduced the development time of new games, further improvements were needed.

One significant economic disadvantageous feature with commercial video wagering gaming units that maintains an artificially high price for the systems in the market is the use of unique hardware interfaces in the various manufactured video gaming systems. The different hardware, the different access codes, the different pin couplings, the different harnesses for coupling of pins, the different functions provided from the various pins, and the other various and different configurations within the systems has prevented any standard from developing within the technical field. This is advantageous to the equipment manufacturer, because the gaming formats for each system are provided exclusively by a single manufacturer, and the entire systems can be readily rendered obsolete, so that the market will have to purchase a complete unit rather than merely replacement software, and aftermarket gaming designers cannot easily provide a single gaming application that can be played on different hardware.

The invention of computerized gaming systems that include a common or "universal" video wagering game controller that can be installed in a broad range of video gaming apparatus without substantial modification to the gaming apparatus controller has made possible the standardization of many components and of corresponding gaming software within gaming systems. Such systems desirably will have functions and features that are specifically tailored to the unique demands of supporting a variety of gaming applications and gaming apparatus types, and doing so in a manner that is efficient, secure, and cost-effective to operate.

What is desired is an architecture and method of providing a gaming-specific platform that features reduced game development time and efficient gaming operation, provides security for the electronic gaming system, and does so in a manner that is cost-effective for gaming software developers, gaming apparatus manufacturers, and gaming apparatus users. An additional advantage is that the use of the platform will speed the review and approval process for gaming applications with the various gaming agencies, bringing the gaming formats and gaming applications to market sooner.

The nature of gaming systems and the stringent controls applied to gaming systems and gaming applications by jurisdictional controls (e.g., the Nevada State Gaming Commission, the New Jersey State Gaming Commission, the Mississippi State Gaming Commission, the California State Gaming Commission, the United Kingdom Gaming Commission, etc.) makes the development of a standard operating system and the ability of the game developers to work with such gaming operating systems unique within the field of computer based designer/developer interactions.

One of the reasons that Microsoft Windows® became the leading operating system throughout the world for personal computers was based upon its business strategy of providing access to Microsoft Windows® on-line to developers using an Application Program Interface (API) through which developers could communicate with the Windows® operating system, without being able to modify the underlying operating system (OS). This enabled Windows® to be supported by a vast network of private developers so that significant amounts of software became available for Windows® while other competing operating systems (e.g., Mac OS, Unix and Linux) had much fewer numbers of software programs available to use with these systems. However, the Microsoft Windows® operating system was not designed to support gaming systems and does not contain the essential software components needed for a gaming jurisdiction approvable operating system or gaming application.

Some game systems (as opposed to gaming systems) also attempted an on-line approach to assisting developers in using proprietary game operating systems for development of games compatible with the game operating system. One such on-line system was Adventurebuilder, which has apparently been removed from active on-line operation, even though its API addressable OS has been archived at http://archive.wustl.edu.languages/smalltalk/Smalltalk/st80_CastleMS-.../CastleMS.s and the entire 195 pages of text can be accessed at that site.

Additionally, U.S. Patent No. 6,181,336 B1 (Chiu et al.) describes a system for providing an integrated, efficient and consistent production environment for the shared development of multimedia productions. Examples of multimedia productions include feature animation films, computerized animation films, interactive video games, interactive movies, and other types of entertainment and/or educational

multimedia works. The development of such multimedia products typically involves heterogeneous and diverse forms of multimedia data. Further, the production tools and equipment that are used to create and edit such diverse multimedia data are in and of themselves diverse and often incompatible with each other. The incompatibility between such development tools can be seen in terms of their methods of operation, operating environments, and the types and/or formats of data on which they operate. The common utilities, methods and services disclosed therein, are used to integrate the diverse world of multimedia productions. By using the common utilities, methods and services provided, diverse multimedia production tools can access, store, and share data in a multiple user production environment in a consistent, safe, efficient and predictable fashion.

SUMMARY OF THE INVENTION

The present invention provides a method for a developer to access a unique gaming operation system that can support a wide variety of gaming applications. The developer can access the operating system through an Application Program Interface (API), respond to input from the developer without alteration of the gaming components stored in the operating system, and then enables the developer, by communication with the operating system, to develop a chip, gaming unit or other software that can be communicationally connected to the operating system to play or execute a gaming application, using the operating system as the primary engine for running the gaming application. The developer is capable of using any desired software system to develop the gaming application (e.g., Windows® 98, Windows® NT, Windows® XT, Mac OS, Unix, Linux, etc.) and still communicate to the gaming operating system through the API. The developed chip or software stored on one or more different media, such as EPROM flash memory, CD ROM, etc. or other software containing the gaming play content of a new gaming format may then be inserted into any gaming box with the host operating system by simply replacing a gaming chip CD ROM, disc or other storage media that has been developed through use of access to the operating system through the API, and that gaming application is

assured of performance and can have a significantly reduced approval time through jurisdictional gaming agencies.

The present invention in various embodiments provides such a method to be practiced on a computerized wagering gaming operating system and apparatus that features a proprietary operating system, such as, for a preferred example, an operating system kernel. The apparatus also may include selected device handlers and system libraries, and have other device handlers that are disabled or removed. The present invention features a system handler application that may be part of the operating system. The system handler loads and executes gaming program objects that are part of the operating system and features nonvolatile storage that facilitates sharing of information between gaming program objects. The system handler of some embodiments further provides an API library of functions callable from the gaming program shared objects, and may in some embodiments facilitate the use of callback functions on change of data stored in nonvolatile storage. A nonvolatile record of the state of the computerized wagering gaming application is stored on the nonvolatile storage, providing protection against loss of the gaming state due to power loss. The system handler application in various embodiments includes a plurality of device handlers, providing an interface to selected hardware and the ability to monitor hardware-related events.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows a computerized wagering game apparatus as may be used to practice an embodiment of the present invention.

Figure 2 shows a more detailed structure of program code executed on a computerized wagering game apparatus, consistent with an embodiment of the present invention.

Figure 3 is a diagram illustrating another exemplary embodiment of a universal gaming system according to the present invention having a universal or open operating system.

Figure 4 is a diagram illustrating one exemplary embodiment of a method of converting a gaming system to a gaming system having an open operating system according to the present invention.

Figure 5 is a diagram illustrating one exemplary embodiment of a set of devices used for interfacing with a device driver or handler in an open operating system in a gaming system according to the present invention.

Figure 6 is a diagram illustrating one exemplary embodiment of a resource manager used in a gaming system according to the present invention.

Figure 7 is a diagram of a table illustrating one exemplary embodiment of a resource file used in a gaming system according to the present invention.

Figure 8 is a diagram illustrating one exemplary embodiment of a cashless gaming system using the universal gaming system according to the present invention.

Figure 9 is a diagram illustrating one exemplary embodiment of configuring a game usable in a gaming system according to the present invention.

Figure 10 is a diagram illustrating another exemplary embodiment of configuring and/or storing a game on a removable media useable in a gaming system according to the present invention.

Figure 11 is a diagram illustrating another exemplary embodiment of a gaming system according to the present invention wherein the game layer is programmable or configurable via a web page at a location remote from the gaming system.

Figure 12 is a diagram illustrating one exemplary embodiment of a web page template used in the gaming system shown in Figure 11.

Figure 13 is a diagram illustrating one exemplary embodiment of nonvolatile memory used in a gaming system according to the present invention, wherein the nonvolatile memory is configured as a RAID system.

Figure 14 is a block diagram that shows the operation of API's between software components.

DETAILED DESCRIPTION OF THE INVENTION

In the following detailed description of sample embodiments of the invention, reference is made to the accompanying drawings that form a part hereof, and in which

is shown by way of illustration specific sample embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that logical, mechanical, electrical, and other changes may be made without departing from the spirit or scope of the present invention. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the invention is defined only by the appended claims. It is essential to an appreciation of the practice of the present invention that the jurisdictional approval requirements and the industry standards of the gaming industry be considered in determination of the skill and technical sophistication of the present technology and invention.

In this document, the term "software component" can refer to any software module or grouping of modules. Under this definition a protocol module could be considered to be a type of software component, as could a complete operating system, or even a piece of an operating system. For the purposes of this document, a "software component" will be considered to be the set of code that an individual operating system provider provides.

The Nevada Gaming Control Board defines a gaming-related software component and uses a simple test to determine if a software subsystem falls under the definition of a gaming device. A primary element of a "Gaming Device" under NRS 463.0155 is a component that must be used remotely or directly in connection with a game (gaming application) and that affects the result of a wager by determining win or loss (based on a probability). Taking into account this definition, the Nevada Gaming Control Board uses a simple test: if an operating system or other software component can be shown to be usable in other fields or applications and the component is not involved in the calculation of wagering win or loss, then it is not a gaming-related piece of software. Those software components that relate to the presentation, decision-making and storage of win/loss information are the ones that are of primary concern to the Commission.

Nevada gaming laws and regulations require that the "Manufacturer" of a gaming device must be licensed. Manufacturer is defined in NRS 463.0172 as one who: "manufactures, assembles, programs or makes modifications to a gaming device or cashless wagering system ... or who designs, controls the design or maintains a copyright over the design of a program which can't be reasonably demonstrated to have any use other than in a gaming device or cashless wagering system."

Therefore, where a gaming device contains software written by multiple vendors, an analysis must be made as to whether a gaming license is required by each vendor depending on whether the component provided by each vendor is gaming related. It may be that a vendor that supplies certain code may not have to be licensed in order for a licensed vendor to include that code on a gaming device. The complications arise when considering the different possibilities that can occur, as considered below.

The concept of trust or authenticity is the basis of all gaming regulations. To have a high degree of confidence in the fairness, integrity and security of a gaming device, it is necessary to be able to trust that the software components really do what they have been tested for and approved to do. There are different ways to establish such trust. Providing source code and other documentation of the software component is one such method. Other methods include public key infrastructure (PKI) to authenticate game code and data, requiring that the physical storage location for the code be on unwritable media, etc. In the case of PKI authentication, it is critical that the software component that provides the authentication service has the highest level of trust. If this is not true, then the purpose is entirely lost. For the purposes of this document, it will help with clarity to refer to the software component that provides authentication services as the operating system, or just OS. This is mainly for readability and understanding, but it is important that in systems that contain multiple software components, an OS/application type-relationship between the components is just one configuration.

Most gaming jurisdictions require devices that contain code that is stored in writable media to have an approved method to verify the integrity of the code that is stored there. Using PKI signatures is a commonly used and accepted method for such authentication. This implies that one software component inherently has a higher trust level than the one being authenticated. In gaming jurisdictions, read-only storage on EPROM *with available source code* has the highest level of trust because the code can be easily verified by spot-checking devices in the field. The code that exists on the EPROM will in turn check the signatures for the code stored on the writable media. Assuming things check out, the device may now proceed with operation. U.S. Patent Nos. 6,149,552; 6,106,396; 6,104,859; and 5,643,086 (The Alcorn Patents) describe various authentication techniques for use in gaming systems. Although authentication and/or encryption systems are essential for commercial computer based gaming products, they need not be present on the system that is provided for access to the developer. The absence of the authentication system at this point in the development procedure may simplify communication and additionally speed up development. The authentication system and the encryption processes attendant thereto may be added into the commercial gaming apparatus without adversely affecting the ability of the developed gaming application or gaming rules chip to operate on the operating system.

The availability of source code mentioned above is extremely important when one considers this hierarchy of trust between software components that has to be strictly enforced in order to not compromise the integrity of the system. Only those software components that have the highest level of trust should be in a position to certify or authenticate other components. Those components that exist on writable media or do not have source code availability automatically have a lower level of trust. In the case of unlicensed vendors of operating systems with no source code available, for example, you have a situation where an untrusted, unproven software component provides authentication to a component on writable media, which is questionable at best. That provides a complication in the development of gaming

application software and hardware to be used on a proprietary operating system, particularly on-line (over the internet) where identification of users may be problematic and control over secondary distribution or redistribution of the operating system and its source codes are problematic. Such uncontrolled distribution could compromise the ultimate security of the gaming apparatus in the casinos, and could lead to a refusal of gaming operators for the proprietary gaming operating system and for gaming applications provided on that operating system.

As indicated above, an API, or *Application Programming Interface*, is a set of methods used to interface from one communicator (e.g., a developer operating on its own computer and operating system) to a distal information component such as a software component (distal meaning over the internet, on-line, off a memory source such as compact disk, floppy disk, connected hard drive, or other information storage media with which the developer can communicate). These methods may be implemented using message passing, function calls using static or dynamic linking, or some other way. The important common function of every API is to isolate the data and low-level functions in one software component from being accessed except through the use of a common set of access methods.

The primary problem with having a number of software components existing on a single system has to do with defining the boundaries between the components. The only way the components can be separated into completely contained pieces is to define an API to which all the components conform. As long as this is the only way in which the pieces interact, the security of the distal information component is satisfactory.

One way of overcoming the delays and difficulties in introducing gaming applications to the industry is by practicing a method within the scope of the present invention. As a first step, a gaming operating system is provided that contains objects that can be used in gaming applications and gaming apparatus (whether video gaming or reel-type gaming apparatus). This gaming operating system would include

functions in a secure computing system (e.g., computer, server, microprocessor, etc.) or memory system (floppy disk, compact disk, optical disk, hard drive, etc.), these functions being useful in gaming apparatus. The developer provides gaming application specific data (that is rules, directions, payout schedules, numbers of rounds, player activity requirements, and the like) to the Application Programming Interface, creates and ultimately compiles the information needed to direct the gaming operating system to execute the functions necessary to play the gaming application, and provide that compiled information to a gaming apparatus with the operating system in a commercial environment to practice the game.

The method comprises assisting in the development of a computer based wagering gaming application with at least the steps of:

providing a gaming operating system comprising a library of at least two software gaming callback functions and/or primary gaming states;

providing an Application Programming Interface enabling communication from a distal intelligence source to the gaming operating system;

communicating with the Application Programming Interface to the functions and/or primary gaming states in the library of the gaming operations;

providing gaming specific data relating to at least one specific gaming application; and

compiling a program specific to at least one gaming application that is compatible with the gaming operating system through the use of the gaming operating system API.

This method could have the compiled program specific to at least one gaming application provided on a storage device. Some gaming applications have multiple games, and/or bonus rounds that can be included on the storage device. The method may be practiced either with or without security features enabled when communicating with the Application Programming Interface is performed. Security features can be added later when the commercial product is introduced or qualified by the regulatory commissions.

Another way of describing the method would be as assisting in the development of a computer based wagering gaming application comprising the steps of:

providing a gaming operating system comprising a library of at least two software gaming elements selected from the group consisting of a random number generator, a game initiation sequence, a value module (e.g., one or more modules providing controls relating to coin changing, coin recognition, currency recognition, credit recognition/storage, ticket recognition/printout, etc.), a bonus module (e.g., bonus, jackpot, additional play, alternative play), a video gaming module (e.g., including actual image files, image sequencing files, clip art files, video storage files [e.g., empty or partial files], color files, etc.), an audio gaming module (sound files, sound sequence files, sound files tied to video events, volume controls, etc.), a jackpot module, a graphics conversion tool, a debugging tool, a pay-out table module, a value-handling module, a power-loss back-up module, a gaming payout history module, a player history module, and a user interaction module (e.g., handle controls, button controls, touch-screen controls, joystick controls, etc.);

providing an Application Programming Interface enabling communication from a distal intelligence source to the gaming operating system;

communicating with the Application Programming Interface to functions and/or primary gaming states in the library of the gaming operating system; and

compiling a program specific to at least one gaming application that is compatible with the gaming operating system.

The method may affect compiling of the program including writing a program that comprises gaming application specific commands that communicate with the gaming operating system. The method may be practiced in conjunction with a user manual with directions on accessing files in the library and is used by using specific commands in the user manual to access specific files or functions in the operating system through the Application Programming Interface.

For those software components which use PKI (public key infrastructure) for authentication of other components in the system, it is desirable to create a new pair of public/private keys for each software release. There are several reasons for this:

- Matching software releases by version
- Simplification of the regulatory approval process
- Barrier to brute-force cracking techniques
- Preventative security measures
- Jurisdictional non-compatibility

"Revving" is the process by which public or private keys are replaced in the OS ROM. Private keys are used to generate signatures and the public key is used to verify the signatures. This ordinarily is done every time that a new version of software is introduced into new gaming jurisdictions or even the same gaming applications to different jurisdictions.

In the software development process, small inconsistencies and incompatibilities will creep into an API as new features are added and changes are made to the internal workings of a software component. This is true especially in an embedded environment where it is not cost effective or space effective to have multiple sections of redundant code. One way to ensure that the devices in the field are using compatible software components is to somehow prevent incompatible versions from co-existing. Revving the public/private key pairs with each release of a trusted software component is one such method.

One significant delay in the introduction of gaming applications to the market has involved the complexity and length of regulatory approval. For example, consider a circumstance where there are hundreds of games deployed with a certain OS. Each of these games runs on a certain version of the software (most likely the latest version) but not necessarily so. When a new version of that OS is released, gaming regulations require that approval is obtained for all possible game configurations which exist in the field. This means that if every version of OS uses the same keys, it would be necessary to test and submit for approval every old game

that existed in the field with old versions, because there would be no mechanism preventing someone from using an old version with new game code.

With a different key pair for every OS version, only new games would ever have to be submitted for approval, since the old games would automatically not work with the new keys and consequently, the new operating system.

If someone wanted to discover a private key and did not have a mathematical way of doing so (for the encryption technique described above, there are no such known methods) they would try to discover the key by guessing. This may be hard to do with a key length of 512 bits, but someone with a lot of computer power might eventually be able to guess the correct key. By using very long keys, this approach has been made as impractical as possible. For this reason, the more impractical it is to guess keys the better, as far as security of keys is concerned. Therefore, if new keys are generated every time a version of software is released, it will be that much more impractical for someone to try to guess the keys. This makes the overall system more secure.

One of the most important aspects to consider with the PKI method of using signatures for the verification of gaming chips is that regulatory agencies base their approval on the confidence that exists in maintaining the secrecy of the private key. Since anyone who might want to cheat the gaming application must modify the game code, if they do not know the private key, they will be unable to generate signatures for the gaming application to work. If a private key is lost, the integrity of every machine in the field which uses that public/private key set is compromised. The only way to correct this compromise in security is to generate a new public/private key pair and upgrade every machine with the new public key. If the same set of keys has been used for every software release, this means the manufacturer must generate a new public/private key pair and upgrade every version of OS in the field. With a different public/private key set for each OS version, only a subset of machines in the field will

have to be upgraded if a key is compromised, which translates to less cost and less disruption to the casino's business.

Another important issue to consider is the fact that some manufacturers may use the same software component in several different jurisdictions. It is desirable to ensure that a gaming application written for one jurisdiction will not operate in a machine in a different jurisdiction. Also, vendors that are licensed in one jurisdiction should not have the keys to produce gaming applications for another jurisdiction. It will be desirable, therefore, to have different sets of keys for different gaming jurisdictions as well as for each software release.

When software components that exist on a system are the property of different vendors, complications arise in the case where one piece is found to have a serious security hole or other bug that causes the regulatory agency to have to disapprove the software. When any software component on a gaming device is disapproved, the gaming device as a whole will be disapproved. There are several issues with this set of circumstances, including at least liability, procedures and unlicensed vendor complications.

The timing of re-submission versus deadlines to remove the disapproved software from the field causes a potential liability issue when a bug fix cannot be found quickly enough. There should most likely be procedures that vendors should follow for tracking software installation and revisions in the field. Without such procedures, it would be impossible to do a coordinated software upgrade if the need arises.

If more than one software component interacts with a third component through the same API, there can be side effects known as couplings. For example, if a software function enables or disables a software feature, then an outside module could call the function to turn the feature on and a second module may then turn the feature

off. Unless the two modules are communicating with each other, there will be problems with the two modules existing on the same system.

In systems where there is an OS / application relationship between software components, there can be a time coupling between all API calls, especially if there is an ability to preemptively multitask the application processes in the system. Because the multitasking can happen at any time, the order in which API calls can happen may change from run to run. This means that the application code needs a layer of error checking to prevent race condition bugs that would otherwise be unnecessary. For this reason, having a single thread of execution for which the order that API calls are made is always the same will automatically have a greater level of trust than will a multitasking OS. This is not to say that a multitasking OS cannot overcome this deficiency by correctly using mutexes, but this obviously is more complicated to test and therefore is harder to trust and obtain approval.

There can also exist couplings between API calls. These couplings are different from the interaction couplings pointed out earlier. These couplings are inherent in the way the API operates. The classic example of this type of coupling is the following API for a voltmeter:

```
Set_range(volts)
Set_sensitivity(volts_per_division)
```

The underlying parameters that are being controlled by both of these API calls are the minimum and maximum voltages that will be sensed by the voltmeter. However, at high voltage range settings, certain sensitivities may be unavailable due to the way the voltmeter senses voltages. This illustrates a functional coupling in an API. The generic way to describe these couplings is that one API call can affect the available settings or range of settings that are available to another API call.

One method of verifying software components is to define a series of regression tests and expected test results for the individual pieces. This method is useful for identifying programming errors and bounds checking problems with the

API implementation, but is less useful in identifying system-level weaknesses which my be inherent in the design.

A front-end code verifier or pre-parser may be written that allows developer code to be scanned before compiling to identify errors in the code as it relates to the gaming operating system API. This type of scanner can be used to find couplings and other errors that a regression tester may not find.

In the past, vendor collaboration by defining a standard API specification in the gaming industry has been difficult and unsuccessful. A good example of this is the various protocol implementations that exist which are not always 100% compatible. With software components co-existing on the same machine, there can be no way around the fact that if something is not 100% compatible with the API specifications, there could easily be bugs introduced which would compromise the integrity of the device, which is clearly unacceptable in any jurisdiction.

For purposes of this disclosure, the following terms have specialized meaning, and are defined below:

"Memory" for purposes of this disclosure is defined as any type of media capable of read/write capability. Examples of memory are RAM, tape, flash memory, disc on chips and floppy disc.

"Shared or Game Program Objects" for purposes of this disclosure are defined as self-contained, functional units of game code that define a particular feature set or sequence of operation for a game. The personality and behavior of a gaming machine of the present invention are defined by the particular set of shared objects called and executed by the operating system. Within a single game, numerous game objects may be dynamically loaded and/or executed.

"Architecture" for purposes of this disclosure is defined as software, hardware or both.

"Dynamic Linking" for purposes of this disclosure is defined as linking at run time.

"API" for purposes of this disclosure is an Application Programming Interface. The API includes a library of functions.

"System Handler" for purposes of this disclosure is defined as a collection of code written to control non-gaming specific device handlers. Examples of device handlers include I/O, sound, video, touch screen, nonvolatile RAM and network devices.

"Gaming Data Variables" for purposes of this disclosure includes at a minimum any or all data needed to reconstruct the gaming state in the event of a power loss.

The present invention comprises various elements to enable the use and installation of a novel gaming operating system that in turn enables more rapid development and deployment of novel gaming games. One element of the invention is a computerized wagering game apparatus comprising:

a computerized game controller operable to control the computerized wagering game having a processor, memory, and nonvolatile storage; and

an operating system comprising: a system handler application which provides gaming related functions and services to game programs; and

an operating system kernel that executes the system handler application. The computerized wagering game apparatus may have the system handler application comprise at least one system selected from the group consisting of a) a plurality of device handlers, b) software having the ability when executed to:

load a gaming program and execute the new gaming program; c) an API with functions callable from the game program; d) an event queue; e) a game personality described in a selected mode; and f) a combination of an event queue that determines the order of execution of each specified device handler; an API having a library of functions; an event queue capable of queuing on a first come, first serve basis; and an event queue capable of queuing using more than one criteria. The computerized wagering game apparatus may have game data modified by gaming program objects that are stored in nonvolatile storage or wherein the system handler and kernel work in communication to hash system handler code and operating system kernel code. By

way of non-limiting examples, the game data modified by gaming program objects may be stored in nonvolatile storage and changing game data in nonvolatile storage causes execution of a corresponding callback function in the system handler application. The computerized wagering game apparatus may have the operating system kernel as a Linux operating system kernel having customized proprietary modules and the kernel has at least one modification wherein each modification is selected from the group consisting of: 1) accessing user level code from ROM, 2) executing from ROM, 3) zeroing out unused RAM, 4) testing and/or hashing the kernel, and 5) disabling selected device handlers. The computerized wagering game apparatus may have the apparatus contain a machine-readable element with machine-readable instructions thereon, the instructions when executed operable to cause the processor to manage at least one gaming program object via a system handler application and to execute a single gaming program object at any one time, wherein gaming program objects are operable to share game data in nonvolatile storage within the processor in the computerized wagering game system.

The computerized wagering game apparatus may have programming direct the gaming apparatus to effect a procedure selected from the group consisting of a) only one gaming program object executes at any one time, b) there are instructions operable when executed to cause a computer to provide functions through an API that comprises a part of the system handler application, and c) when instructions are executed, the instructions are operable to store game data in nonvolatile storage, such that the state of the computerized wagering game system is maintained when the machine loses power.

A method of assisting in the development of a computer based wagering gaming application can utilize any of the apparatus described herein by the steps of:

providing a gaming operating system comprising a library of at least two software gaming callback functions and/or primary gaming states;

providing an Application Programming Interface enabling communication from a distal intelligence source to the gaming operating system;

communicating with the Application Programming Interface to the functions and/or primary gaming states in the library of the gaming operating system by providing a Makefile or other procedure for building a gaming application, and a configuration file for running the gaming operation system on a proximal computing system;

providing gaming specific data relating to at least one specific gaming application; and

compiling a program specific to at least one gaming application that is compatible with the gaming operating system.

This method of assisting in the development of a computer based wagering gaming application may also comprise using a library of at least two software gaming elements comprising gaming elements selected from the group consisting of random number generator, game initiation sequence, bonus module, video gaming module, audio gaming module, jackpot module, graphics conversion tool, debugging tool, pay-out table module, value-handling module, power-loss recovery module, gaming payout history module, player history module, and user interaction module. Also, the process may have public and/or private authentication keys revved and different public and/or private authentication keys are provided to each of at least two different legal jurisdictions.

A method of managing data in a computerized wagering game apparatus as described herein can be practiced via a system handler application in a method of loading a shared object, executing the shared object, and accessing and storing game data in nonvolatile storage. This method may have further steps of a) unloading the first program object, and loading a second program object or b) executing a corresponding callback function upon alteration of game data in nonvolatile storage.

The present invention also includes a machine-readable memory storage element with instructions thereon, the instructions when executed operable to cause a computer to: load a first program shared object, execute a first program shared object, store gaming data in nonvolatile storage, such that a second program object later

loaded can access gaming data variables in nonvolatile storage, unload the first program shared object from system memory, and load the second program shared object to system memory so that the second program shared object is accessible to the computer as instructions. This machine-readable memory storage element may have additional instructions operable when executed to cause a computer to perform a task selected from the group consisting of a) executing a corresponding callback function upon alteration of game data in the nonvolatile storage; and b) managing events via the system handler application.

Another aspect of the present invention includes a universal operating system stored in a memory storage component that may be operatively inserted along with game identity data into an electronic or electromechanical gaming device having ancillary functions so that the gaming device can effect play of the game provided in the game identity data. The operating system will control at least one ancillary function selected from the group consisting of coin acceptance, credit acceptance, currency acceptance and boot up, the gaming device having at least one system handler application, and the operating system comprising a system handler and an operating system kernel. This operating system may also have at least one of a plurality of APIs, an operating system kernel customized for gaming purposes, and an event queue, or a system handler having a plurality of device handlers or the operating system controls a networked on-line system or control a progressive meter. The operating system may also have a kernel customized for gaming purposes utilizing a method of operation selected from the group consisting of: 1) accessing user level code from ROM, 2) executing from ROM, 3) zero out unused RAM, 4) test and/or hash the kernel, and 5) disabling selected device handlers.

Another method within the scope of the invention can be generally described as a) customizing an operating system kernel and b) providing the customized kernel of the operating system into a gaming apparatus, at least one customization being effected to obtain functionality of the gaming apparatus, the customization being a kernel modification for a process selected from the group consisting of:

1) accessing user level code from ROM;
2) executing user level code from ROM;
3) zeroing out unused RAM;
4) testing and/or hashing the kernel; and
5) disabling selected device handlers.

Another method within the scope of the invention can be generally described as converting a first game that operates on a first gaming system so that the game operates on a universal gaming system, the method comprising: removing a first game operating system from the gaming system, the first game operating system including hardware and software; installing the universal gaming system in place of the game operating system, the universal gaming system including a game program layer, an open operating system, and a game controller for running the game program layer on the open operating system; providing functional interfaces between the universal gaming system and game devices; and installing a second game specific program in the game program layer configured to operate with the open operating system. This method may have at least one step selected from the group consisting of:

a) providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the game devices via the system application handler;

b) configuring the system handler application to include one or more device handlers for interfacing with the game devices, wherein at least one of the device handlers operates as a protocol manager between the games device and the open operating system;

c) providing the open operating system to include an operating system kernel that executes the system handler application; and

d) providing the game program layer with at least one gaming program object.

This method may have the at least one gaming program object specific to a type of game played on the universal gaming system. The method may also have at least one step selected from the group consisting of :

changing a type of game played on the universal gaming system by changing game program objects;

configuring the game program layer to operate the game as a slot machine;

operating the slot machine as a mechanical reel-based slot machine; and

configuring the open operating system to include a resource manager for mapping game specific resources.

This method may include mapping game specific resources by parsing a configuration file, mapping operating system resources based on the configuration file, and storing the resource map in memory. This mapping of the operating system resources may be based on the configuration file includes mapping input/output lines to system resources. The method may enable converting the first gaming system from a cash accepting gaming system to a cashless gaming system, the method including providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the game devices accomplished via the system application handler, and configuring the system handler application to include one or more device handlers for interfacing with the game devices, the configuring including installing a card reader device handler, and installing a card reader in communication with the card reader device handler, and optionally including configuring the system handler application to include a ticket printer device handler; and installing a ticket printer in communication with the ticket printer device handler. This method may be practiced, by way of a non-limiting example on a slot machine game operating system that is removed from the first gaming system and where the functional interfaces are between the universal gaming system and slot machine game devices. This method may also perform at least one step selected from the group consisting of: a) providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the slot machine game devices via the system application handler; b) configuring the system handler application to include one or more device handlers for interfacing with the

slot machine game devices, wherein at least one of the device handlers operates as a protocol manager between the slot machine games device and the open operating system; c) configuring an I/O device handler to interface with slot machine input devices and slot machine output devices; d) providing slot machine input devices that include a mechanical arm, button acceptor and coin acceptor; and e) providing the slot machine with output devices inclusive of slot machine reels, credit displays, and speakers. The method may act to convert the mechanical reel slot machine game having only cash, token, credit balance and currency acceptance capability to a cashless gaming system via the system handler application, the converting including providing a card reader device handler, and installing a card reader in communication with the card reader device handler and optionally providing a ticket printer device handler, and installing a ticket printer in communication with the ticket printer device handler.

Another aspect of the method of the present invention is a method of configuring a game program layer for a universal gaming system that is configured for a game program layer and an open operating system, the method comprising: configuring the game program layer on a computer remote from a first non-universal gaming system; and downloading the game program layer into the universal gaming system and performing at least one sequence comprising:

a) defining a game template; and configuring the game program layer using the game template;

b) storing the game program on a removable media card; and

c) providing removable media as flash memory.

This method may be practiced, for example, where the game program is stored on a removable media card and the removable media card is plugged into the gaming system, and then running the game program layer via the open operating system from the removable media card. This method may also have an additional step of preparing the game program layer for authentication by plugging the removable media card into an authenticating system. This method may be performed, in a non-limiting example, as a network based method of providing a game program layer for a universal gaming

system configured for remote operation using an open operating system, the method including defining a user interface to communicate between the remote computer and the universal operating system. For example, the game program layer is configured to use user interface remote from the gaming system or via a web page template at the user interface.

The present invention may also comprise a gaming system suitable for use in a casino comprising: a game controller configured to operate the gaming system; and a first nonvolatile memory and a second nonvolatile memory for storing critical gaming information, wherein the first nonvolatile memory and the second nonvolatile memory are configured to communicate with the game controller as a gaming RAID system for redundant storage of critical gaming information. RAID not defined in text, the gaming system enabling redundant NVRAM storage to be replaceable while operating power for the system is on.

The present invention includes a method of accessing a computerized gaming operating system by a method and apparatus. The operating system has novel gaming-specific features that improve security, make development of game code more efficient, and do so using an apparatus and software methods that are cost-effective and efficient. The present invention also reduces the amount of effort required to evaluate and review new game designs by gaming regulators, because the amount of code to be reviewed for each game is reduced by at least as much as 40%, preferably at least 50%, more preferably at least 60% or even at least 70%, and as much as 80% or more over known, machine-specific architecture that one skilled in the art might wish to insert into gaming systems. That is, in the practice of the present invention, rather than having every line of code or software screened, certain software is essentially 'pre-approved' by previous inspection and only game code additions (and the like) need to be reviewed for approval. The invention provides, in various embodiments, features such as a nonvolatile memory for storing gaming application variables and game state information, provides a shared object architecture that allows individual game objects to be loaded and to call common functions provided by a

system handler application, and in one embodiment provides a custom operating system kernel that has selected device handlers disabled.

Incorporated by reference in this description is the Shuffle Master Gaming, Game Operating System "SGOS" Developer's Manual, revised May 2001 comprising 175 pages that are attached hereto and incorporated herein as part of this specification. This Developer's Manual has not been published, but has been provided under limited access under confidentiality agreements with potential developers, and does not constitute prior art. No commercial products have been introduced using this manual or the development procedures and systems of the present invention as of 10 September 2001.

Figure 1 shows an exemplary gaming system 100, illustrating a variety of components typically found in gaming systems and how they may be used in accordance with the present invention. User interface devices in this gaming system include push buttons 101, joystick 102, and pull arm 103. Credit for wagering may be established via coin or token slot 104, a device 105 such as a bill receiver or card reader, or any other credit input device. A card reader 105 may also provide the ability to record credit information on a user's card when the user has completed gaming, or credit may be returned via a coin tray 106 or other credit return device. Information is provided to the user by devices such as video screen 107, which may be a cathode ray tube (CRT), liquid crystal display (LCD) panel, plasma display, light-emitting diode (LED) display, mechanical reels or wheels or other display device that produces a visual image under control of the computerized game controller. Also, buttons 101 may be lighted to indicate what buttons may be used to provide valid input to the game system at any point in the game. Still other lights or other visual indicators may be provided to indicate game information or for other purposes such as to attract the attention of prospective game users. Sound is provided via speakers 108, and also may be used to indicate game status, to attract prospective game users, to provide player instructions or for other purposes, under the control of the computerized game controller.

The gaming system 100 further comprises a computerized game controller 111 and I/O interface 112, connected via a wiring harness 113. The universal game controller 111 need not have its software or hardware designed to conform to the interface requirements of various gaming system user interface assemblies, but can be designed once and can control various gaming systems via the use of machine-specific I/O interfaces 112 designed to properly interface an input and/or output of the universal computerized game controller to the harness assemblies found within the various gaming systems.

In some embodiments, the universal game controller 111 is a standard IBM Personal Computer-compatible (PC compatible) computer. Still other embodiments of a universal game controller comprise general purpose computer systems such as embedded controller boards or modular computer systems. Examples of such embodiments include a PC compatible computer with a PC/104 bus that is an example of a modular computer system that features a compact size and low power consumption while retaining PC software and hardware compatibility. The universal game controller 111 provides all functions necessary to implement a wide variety of games by loading various program code on the universal controller, thereby providing a common platform for game development and delivery to customers for use in a variety of gaming systems. Other universal computerized game controllers consistent with the present invention may include any general-purpose computers that are capable of supporting a variety of gaming system software, such as universal controllers optimized for cost effectiveness in gaming applications or that contain other special-purpose elements yet retain the ability to load and execute a variety of gaming software. Examples of special purpose elements include elements that are heat resistant and are designed to operate under less than optimal environments that might contain substances such as dust, smoke, heat and moisture. Special purpose elements are also more reliable when used 24 hours per day, as is the case with most gaming applications.

The computerized game controller of some embodiments of the present invention is a computer running an operating system with a gaming application-specific kernel. In alternative or further embodiments, a game engine layer of code executes within a non-application specific kernel, providing common game functionality. The gaming program shared object in such embodiments is therefore only a fraction of the total code, and relies on the game engine layer and operating system kernel to provide a library of gaming functions. A preferred operating system kernel is the public domain Linux 2.2 kernel available on the Internet. Still other embodiments will have various levels of application code, ranging from embodiments containing several layers of game-specific code to a single-layer of game software running without an operating system or kernel but providing its own computer system management capability.

Figure 2 illustrates the structure of one exemplary embodiment of the invention, as may be practiced on a computerized gaming system such as that of Figure 1. The invention includes an operating system 300, including an operating system kernel 201 and a system handler application 202. An operating system kernel 201 is first executed, after which a system handler application 202 is loaded and executed. The system handler application in some embodiments may load a gaming program shared object 203, and may initialize the game based on gaming data variables stored in nonvolatile storage 204. In some embodiments, the gaming data variables are mapped using a Game.State data file 205, which reflects the data stored in nonvolatile storage 204. The nonvolatile RAM (NV-RAM) according to the invention has read/write capability. The gaming program object in some embodiments calls separate API functions 206, such as sound functions that enable the gaming apparatus to produce sound effects and music.

The OS kernel 201 in some embodiments may be a Linux kernel, but in alternate embodiments may be any other operating system providing a similar function. The Linux 2.2 operating system kernel in some further embodiments may be modified for adaptation to gaming architecture. Modifications may comprise

erasing or removing selected code from the kernel, modifying code within the kernel, adding code to the kernel or performing any other action that renders certain device handler code inoperable in normal kernel operation. By modifying the kernel in some embodiments of the invention, the function of the computerized gaming apparatus can be enhanced by incorporating security features, for example. In one embodiment, all modifications to the kernel are of the form of proprietary kernel modules loadable at run-time.

In one embodiment, the system is used to execute user level code out of ROM. The use of the Linux operating system lends itself to this application because the source code is readily available. Other operating systems such as Windows and DOS are other suitable operating systems.

Embodiments of the invention include hard real time code 310 beneath the kernel providing real time response such as fast response time to interrupts. The hard real time code 310 is part of the operating system in one embodiment.

In one embodiment of the invention, all user interface peripherals such as keyboards, joysticks and the like are not connected to the architecture so that the operating system and shared objects retain exclusive control over the gaming machine. In another embodiment, selected device handlers are disabled so that the use of a keyboard, for example, is not possible. It is more desirable to retain this functionality so that user peripherals can be attached to service the machine. It might also be desirable to attach additional user peripherals such as tracking balls, light guns, light pens and the like.

In another embodiment, the kernel is further modified to zero out all unused RAM. This function eliminates code that has been inserted unintentionally, such as through a Trojan horse, for example.

In one embodiment, the kernel and operating system are modified to hash the system handler and shared object or gaming program object code or both, and to hash

the kernel code itself. These functions in different embodiments are performed continuously, or at a predetermined frequency.

The system handler application is loaded and executed after loading the operating system, and manages the various gaming program shared objects. In further embodiments, the system handler application provides a user Application Program Interface (API) 206 that includes a library of gaming functions used by one or more of the shared objects 210. For example, the API in one embodiment includes functions that control graphics, such as color, screen commands, font settings, character strings, 3-D effects, etc. The device handler callbacks 210 are preferably handled by an event queue 320. The event queue schedules the event handlers in sequence. The shared object 203 calls the APIs 206 in one embodiment. The system handler application 202 in various embodiments also manages writing of data variables in the "game.state" file 205 into the nonvolatile storage 204, and further manages calling any callback functions associated with each data variable changed.

The system handler 202 application of some embodiments may manage the gaming program shared objects by loading a single object at a time and executing the object. When another object needs to be loaded and executed, the current object may remain loaded or can be unloaded and the new object loaded in its place before the new object is executed. The various shared objects can pass data between objects by storing the data in nonvolatile storage 204. For example, a "game.so" file may be a gaming program object file that is loaded and executed to provide operation of a feature set of a computerized wagering game, while a "bonus.so" gaming program object file is loaded and executed to provide a feature set of the bonus segment of play. Upon changing from normal game operation to bonus, the bonus.so is loaded and executed upon loading. Because the relevant data used by each gaming program object file in this example is stored in nonvolatile storage 204, the data may be accessed as needed by whatever gaming program object is currently loaded and executing.

The system handler application in some embodiments provides an API that comprises a library of gaming functions, enabling both easy and controlled access to various commonly used functions of the gaming system. Providing a payout in the event of a winning round of game play, for example, may be accomplished via a payout function that provides the application designer's only access to the hardware that pays out credit or money. Restrictions on the payout function, such as automatically reducing credits stored in nonvolatile storage each time a payout is made, may be employed in some embodiments of the invention to ensure proper and secure management of credits by the computerized gaming system. The functions of the API may be provided by the developer as part of the system handler application, and may be a part of the software provided in the system handler application package. The API functions may be updated as needed by the provider of the system handler application to provide new gaming functions as desired. In some embodiments, the API may simply provide functions that are commonly needed in gaming, such as computation of odds or random numbers, an interface to peripheral devices, or management of cards, reels, video output or other similar functions.

The system handler application 202 in various embodiments also comprises a plurality of device handlers 210 that monitor for various events and provide a software interface to various hardware devices. For example, some embodiments of the invention have handlers for nonvolatile memory 212, one or more I/O devices 214, a graphics engine 216, a sound device 218, or a touch screen 220. Also, gaming-specific devices such as a pull arm, credit receiving device or credit payout device may be handled via a device handler 222. Other peripheral devices may be handled with similar device handlers, and are to be considered within the scope of the invention. In one embodiment, the device handlers are separated into two types. The two types are: soft real time 210A and regular device handlers 210B. The two types of device handlers operate differently. The soft real time handler 210A constantly runs and the other handler 210B runs in response to the occurrence of events.

The nonvolatile storage 204 used to store data variables may be a file on a hard disc, may be nonvolatile memory, or may be any other storage device that does not lose the data stored thereon upon loss of power. In one embodiment the nonvolatile storage is battery-backed RAM. In another embodiment, the non-volatile storage is flash memory. The nonvolatile storage in some embodiments may be encrypted to ensure that the data variables stored therein cannot be corrupted. Some embodiments may further include a game.state file 205, which provides a look-up table for the game data stored in nonvolatile storage 204. The game.state file is typically parsed prior to execution of the shared object file. The operating system creates a map of NVRAM by parsing the game.state file. The look-up table is stored in RAM. This look-up table is used to access and modify game data that resides in NVRAM 204. This game data can also be stored on other types of memory.

In some embodiments, a duplicate copy of the game data stored in NVRAM 204 resides at another location in the NVRAM memory. In another embodiment, a duplicate copy of the game data is copied to another storage device. In yet another embodiment, two copies of the game data reside on the NVRAM and a third copy resides on a separate storage device. In yet another embodiment, three copies of the game data reside in memory. Extra copies of the game data are required by gaming regulations in some jurisdictions.

Data written to the game state device must also be written to the nonvolatile storage device, unless the game state data device is also nonvolatile, to ensure that the data stored is not lost in the event of a power loss. For example, a hard disc in one embodiment stores a file that contains an unencrypted and nonvolatile record of the encrypted data variables in nonvolatile storage flash programmable memory (not shown). Data variables written in the course of game operation may be encrypted and stored in the nonvolatile storage 204, upon normal shutdown. Loss of power leaves a valid copy of the most recent data variables in the non-volatile storage.

In an alternate embodiment, a game state device 205 such as a game.state file stored on a hard disc drive provides variable names or tags and corresponding locations or order in nonvolatile storage 204, in effect, providing a variable map of the nonvolatile storage. In one such embodiment, the nonvolatile storage may then be accessed using the data in the game state file 205, which permits access to the variable name associated with a particular nonvolatile storage location. Such a method permits access to and handling of data stored in nonvolatile storage using variable names stored in the game state file 205, allowing use of a generic nonvolatile storage driver where the contents of the nonvolatile storage are game-specific. Other configurations of nonvolatile storage such as a single nonvolatile storage are also contemplated, and are to be considered within the scope of the invention.

Callback functions that are managed in some embodiments by the system handler application 202 may be triggered by changing variables stored in NVRAM 204. For each variable, a corresponding function may be called that performs an action in response to the changed variable. For example, every change to a "credits" variable in some embodiments calls a "display_credits" function that updates the credits as displayed to the user on a video screen. The callback function may be a function provided by the current gaming program shared object or can be called by a different gaming program object.

The gaming program's shared objects in some embodiments of the invention define the personality and function of the game. Program objects provide different game functions, such as bookkeeping, game operation, game setup and configuration functions, bonus displays and other functions as necessary. The gaming program objects in some embodiments of the invention are loaded and executed one at a time, and share data only through NVRAM 204 or another game data storage device. The previous example of unloading a game.so gaming program object and replacing it with a bonus.so file to perform bonus functions is an example of such use of multiple gaming program shared objects.

Each gaming program object may require certain game data to be present in NVRAM 204, and to be usable from within the executing gaming program shared object 203. The game data may include meter information for bookkeeping, data to recreate game on power loss, game history, currency history, credit information, and ticket printing history, for example.

The operating system of the present application is not limited to use in gaming machines. It is the shared object library rather than the operating system itself that defines the personality and character of the game. The operating system of the present invention can be used with other types of shared object libraries for other purposes.

For example, the operating system of the present invention can be used to control networked on-line systems such as progressive controllers and player tracking systems. The operating system could also be used for kiosk displays or for creating "picture in picture" features in gaming machines. A gaming machine could be configured so that a video slot player could place a bet in the sports book, then watch the sporting event in the "picture in picture" feature while playing his favorite slot game.

The present invention provides a computerized gaming apparatus and method that provides a gaming-specific platform that features reduced game development time and efficient game operation via the use of a system handler application that can manage independent gaming program objects and gaming-specific API, provides game functionality to the operating system kernel, provides security for the electronic gaming system via the nonvolatile storage and other security features of the system, and does so in an efficient manner that makes development of new software games relatively easy. Production and management of a gaming apparatus is also simplified, due to the system handler application API library of gaming functions and common development platform provided by the invention.

Figure 3 is a diagram illustrating one exemplary embodiment of a gaming system 400 according to the present invention including universal operating system 300. As previously described herein, game layer 402 includes gaming program shared objects 203 which are specific to the type of game being played on gaming system 400. Exemplary game objects or modules include paytable.so 406, help.so 408 and game.so 410. Game layer 402 also includes other game specific independent modules 412. Game engine 404 provides an interface between game layer 402 and universal operating system 300. The game engine 404 provides an additional application programming interface to the game layer application. The game engine 404 automates core event handling for communicating with the game operating system 300, and which are not configurable via the specific game layer game code. The game engine 404 also provides housekeeping and game state machine functions. The game layer objects 203 and/or modules 406, 408, 410 may also directly call user API 206.

As previously described herein universal operating system 300 is an open operating system which allows for conversion of the gaming system 400 into different types of games, and also allows for future expandability and upgrading of associated hardware in the gaming system 400 due to its open architecture operating system.

In operating system 300, device handlers 210 provide the interface between the operating system 300 and external gaming system input and output devices, such as a button panel, bill acceptor, coin acceptor, mechanical arm, reels, speaker, tower lights, etc. Each device handler 210 is autonomous to the other. The device handlers or drivers 210 operate as protocol managers, which receive information from a gaming system device (typically in the gaming system device protocol) and convert the information to a common open operating system protocol usable by operating system 300. Similarly, the device drivers or handlers 210 receive information from the open operating system and convert the information to a gaming device specific protocol. The specific device handlers or drivers used are dependent upon what game you are using, and may be loadable or unloadable as independent, separate objects or

modules. The exemplary embodiment shown includes total I/O device handler 414, sound device handler 416, serial device handler 418, graphics device handler 420, memory manager device handler 422, NVRAM device handler 424, protocols device handler 426, resource manager device handler 428 and network device handler 430. Other suitable device handlers for adapting the operating system 300 to other gaming systems will become apparent to one skilled in the art after reading the present application.

Figure 4 is a diagram illustrating one exemplary embodiment of a method of converting an existing gaming operating system to a gaming system 400 having an open operating system 300 according to the present invention. The gaming system 400 according to the present invention is suitable for converting both video based gaming systems and also electrical/mechanical based operating system, such as a mechanical reel based slot machine, and combinations of the two in a unit (by way of a non-limiting example, where one system is an underlying game and the other system is a bonus, jackpot or contemporaneous game). Once the existing game operating system has been changed over to a universal gaming system 400 having a universal operating system 300 according to the present invention, the type of game itself may be changed via changing out the game specific code in the game layer 402.

At 450, the existing game operating system is removed from the game. The existing game operating system is typically a proprietary operating platform consisting of computer hardware and software which is specific to the game being changed out. At 452, a universal gaming system 402 including an open operating system 300 is installed in the game. At 454, functional interfaces are provided between the open operating system and the existing gaming system devices. At 456, a game specific program (i.e., game layer 402) is installed in the universal gaming system. The game specific program is configured to communicate with the open operating system 300.

In one exemplary embodiment, the gaming system according to the present invention is used in a mechanical reel-based slot machine, either in a new slot machine or in converting an existing slot machine to an open operating system according to the present invention. Exemplary conventional reel-based slot machines include an IGT S-plus slot machine or a Bally™ slot machine.

Figure 5 is a diagram illustrating one exemplary embodiment of I/O devices which must be functionally interfaced within adopting gaming system 402 to a reel-based game. The exemplary embodiment shown includes devices which interface with a digital I/O device driver. In one embodiment, input devices 462 includes a button panel device 466, a mechanical arm device 468, a bill acceptor device 470, and a coin acceptor device 472. Each of the input devices 462 receives information from the specific game devices and provides the information to the gaming system 400 via the I/O device driver.

Output devices 464 receive information from operating system 300 which provides an output via the I/O device driver to gaming devices 464. In the example shown, output devices 464 include reels device 474 which receives an output to the stepper motors controlling the reels. Credit displays device 476 which receives an output to the LED driven credit displays. Speaker device 478 which receives a sound output to the game speakers. On-line system protocol devices 480 are communication interfaces between the open operating system 300 and the game on-line system. Tower light devices 482 receive an interface between the open operating system 300 and the game tower lights.

Figure 6 is a diagram illustrating one exemplary embodiment of a resource manager used in a gaming system according to the present invention. The resource manager 500 is a software module which receives a resource configuration file 502 and stores it in memory 504. In one aspect, memory 504 is stored in nonvolatile memory, which in one embodiment is flash memory. The resource manager parses the resource configuration file and stores individual resources in memory for fast recall.

In one embodiment, the resource manager 500 stores the file 502 in the form of a lookup table. In one preferred embodiment, the resource manager reads the configuration files at startup, parses the configuration files and stores them in memory 504. The resource manager file 506 may then be accessed by the rest of the operating system 300 software applications. The resource manager operates to map digital I/O lines, com ports, game specific resources, kernel modules to load, etc.

Figure 7 is a diagram of a table illustrating one exemplary embodiment of a portion of a resource file 506 according to the present invention. The resource manager 500 operates to map the input/output (I/O) line to the operating system resources. Instead of changing pin locations for different games, the resource manger provides for mapping of I/O lines via software. In one aspect, 64, I/O (X8) lines are mapped to the various operating system resources. In one aspect, the I/O line at PIN# 1 510 is mapped to resource R20 512; and PIN# 2 514 is mapped to resource R3 516; PIN# 3 518 is mapped to resource R38 520; PIN# 4 522 is mapped to resource R10 524; PIN# 5 526 is mapped to resource R11 528; PIN# 6 530 is mapped to resource R12 532; PIN# 7 534 is mapped to resource R13 536; and PIN# N 538 is mapped to resource R51 540, etc.

The gaming system 400 according to the present invention is adaptable for use as a cashless gaming system. As such, it is useable for converting existing coin-based or token-based gaming systems into a cashless gaming system.

Figure 8 is a diagram illustrating one exemplary embodiment of converting cash, coin, or token-based gaming system to a cashless gaming system using the universal gaming system 400 according to the present invention. References also made to Figures 1-7 previously described herein. A card reader or coupon acceptor 550 and ticket printer 552 are added to the game. The card reader 550 and ticket printer 552 are easily adaptable to interface with the gaming system 400 according to the present invention. In particular, card reader device driver 554 is added to open operating system 300 to enable card reader 550 to communicate with the operating

system. Similarly, a ticket printer device driver 556 is added to the operating system 300 in order to allow ticket printer 552 to communicate with the operating system. For example, an existing cash-based reel slot machine can be converted according to the present invention to a cashless gaming system. The card reader 550 can operate to read credit cards, magnetic strip based cards, or accept coupons which includes credits such as promotional gaming credits received from a casino. The card or coupons may be obtainable from a central or kiosk location. Once play is complete on the gaming system 400, the ticket printer 556 is operable to print a ticket representative of the amount of credits or money won on the gaming system. The ticket 560 may then be used as a card or coupon in another gaming system, or alternatively, may be turned in at a kiosk or central location for money.

Figure 9 is a diagram illustrating another exemplary embodiment of the gaming system 400 according to the present invention. Due to the open operating system 300, game layer 402 may be configurable remote from the gaming system 400, such as on a remote computer or laptop computer 580. Game layer 402 is constructed into game objects or modules 302. As such, templates for specific types of games are configured to allow a game programmer to specify game specific configurable options from a remote computer 580. In another aspect, game specific modules are created on the remote computer 580. The game layer is then assembled and transferred into memory 582. In one aspect, memory 582 is nonvolatile memory located in the gaming system 400. In one aspect, the nonvolatile memory is flash memory. In one exemplary embodiment, the flash memory is a "Disk on a Chip", wherein the game layer 402 is downloaded from the remote computer 580 onto the disk on a chip 582.

Figure 10 is a diagram illustrating another exemplary embodiment of programming and/or configuring a game layer at a location remote from the gaming system 400. In this embodiment, game layer 402 is programmed or configured on remote computer 580. After completion of configuring and/or programming game layer 402, the game layer 402 is transferred via remote computer 580 to a removable

media 584. In one preferred embodiment, the removable media is a flash memory card, and more preferably is a CompactFlash™ card. In one aspect, the flash memory card plugs into remote computer 580 via a PCMCIA slot. Suitable flash memory cards (i.e., a CompactFlash™ card) are commercially available from memory manufacturers, including SanDisk and Kingston.

The removable media 584 is removed from remote computer 580 and inserted in gaming system 400. In one aspect, removable media 584 can be "hot-inserted" directly into the controller board of gaming system 400. The removable media 584 contains game layer 402 including the game specific code and program files. As such, removable media 584 remains inserted into gaming system 400 during operation of the gaming system. In an alternative embodiment, the game layer 402 can be transferred (e.g., via a memory download) from removable media 584 to memory inside of gaming system 400.

In one embodiment, the removable media 584 is maintained in gaming system 400 during operation of the gaming system. As such, the gaming system program files may be verified for authenticity by gaming officials by simply removing the removable media 584 and inserting it in a computer or controller used for verifying/authenticating game code, indicated at 586.

Figure 11 is another exemplary embodiment of a gaming system according to the present invention wherein the game layer is programmable or configurable at a location remote from the gaming system 400. In this embodiment, game layer 402 is configurable over a network based communication system. In one embodiment, network based system 600 includes a user interface 602, network or network communication link 604, and controller 606. Controller 606 is configured to communicate with user 610 via network 604. In particular, centralized controller 606 includes web server 612. User 610 accesses web server 612 via user interface 602, and downloads a web page suitable for configuring a game layer. In one aspect, the web page includes game specific game templates 608, which are utilized for inputting

specific user configurations for individual games. Once a game template 608 has been configured, the game template is transferred to controller 606 via network 604. Controller 606 receives the configuration information associated with game template 608 and assembles game layer or program 402 using the configuration information. The game layer or program 402 can now be downloaded into memory in gaming system 400 for use by gaming system 400 including the game specific configurable options determined by user 610.

The system 600 also allows other user interfaces 614 for configuring games which may be assembled by controller 606 for use in other gaming systems. Alternatively, other user interface 614 may be representative of a gaming official checking the game 402 and authorizing use of the game 402 and gaming system 400. As such, the game layer 402 may be transferred to the gaming system 400 via controller 606, or via a communication link with user interface 614, which may be a direct connection or may be a network. Alternatively, game layer 402 may be transferred from controller 606 or user interface 614 by putting it on a flash memory device (e.g., Disk on a Chip or CompactFlash card) and physically inserted into gaming system 400.

Network 604, as used herein, is designed to include an internet network (e.g., the Internet), intranet network, or other high-speed communication system. In one preferred embodiment, network 604 is the Internet. While the exemplary embodiment and this detailed description refers to the use of web pages on the Internet network, it is understood that the use of other communication networks or next generation communication networks or a combination of communication networks (e.g., and intranet and the Internet) are within the scope of the present invention. The configuration information received from user interface 602 can be assembled into game layer 402 using hardware via a microprocessor, programmable logic, or state machine, in firmware, and in software within a given device. In one aspect, at least a portion of the software programming is web-based and written in HTML and/or JAVA programming languages, including links to the web pages for data collection,

and each of the main components communicate via network 604 using a communication bus protocol. For example, the present invention may or may not use a TC/IP protocol suite for data transport. Other programming languages and communication bus protocols suitable for use with the system 600 according to the present invention will become apparent to those skilled in the art after reading the present application.

Figure 12 is a diagram illustrating one exemplary embodiment of web page game templates used in the system shown in Figure 11. Template 1 is shown at 622 and Template 2 is shown at 624. In one embodiment, upon accessing controller 606 via user interface 602, user 610 selects a game type that the user 610 would like to either program or configure. An example of a game type is a poker template. Based on the game type 626, a template appears at user interface 602 for that game type which allows the user to specify game configurable options, indicated at 628. The controller then operates to assemble the game layer or game programs 402 based on the information received via the game template. The configurable options may include any type of game specific configurable options, such as game colors, game sound, percentage payouts, game rules, game options, etc.

Figure 13 is a diagram illustrating one exemplary embodiment of nonvolatile RAM used in a gaming system 400 according to the present invention, wherein the nonvolatile RAM is configured as a redundant memory system. In one exemplary embodiment shown, the nonvolatile RAM is configured as a RAID system. In the hard disk drive industry, RAID (short for redundant array of independent disks) systems employ two or more disk drives in combination for improved disk drive fault tolerance and disk drive performance. RAID systems stripe a user's data across multiple hard disks. When accessing data, the RAID system allows all of the hard disks to work at the same time, providing increase in speed and reliability.

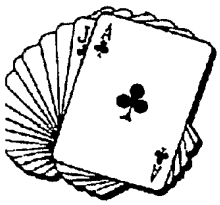
A RAID system configuration as defined by different RAID levels. The different RAID levels range from LEVEL 0 which provides data striping (spreading out of data blocks of each file across multiple hard disks) resulting in improved disk drive speed and performance but no redundancy. RAID LEVEL 1 provides disk mirroring, resulting in 100 percent redundancy of data through mirrored pairs of hard

disks (i.e., identical blocks of data written to two hard disks). Other drive RAID levels provide variations of data striping and disk mirroring, and also provide improved error correction for increased performance and fault tolerance.

In Figure 13, one exemplary embodiment of RAID data storage system used in a gaming system 400 according to the present invention is generally shown at 630. The RAID storage system 630 includes a controller or control system 632 and multiple nonvolatile RAM data storage units, indicated as RAMA 634 and RAMB 636. In one aspect, RAMA 634 and RAMB 636 each include a backup power system PWR 638 and PWR 640. In one aspect, backup power systems PWR 638 and PWR 640 are battery backup systems. RAMA 634 and RAMB 636 are configured to communicate with control system 632 as a redundant array of storage devices. Preferably, nonvolatile memory RAMA 634 and nonvolatile memory RAMB 636 are configured similar to a RAID level configuration used in the disk drive industry (i.e., as a "mirrored pair"). Nonvolatile memory RAMA 634 and nonvolatile memory RAMB 636 communicate with control system 632 via communication bus 638, using a communication bus protocol. One exemplary embodiment of a communication bus suitable for use as communication bus 638 is an industry standard ATA or uniform serial bus (USB) communication bus. Control system 632 includes a microprocessor based data processing system or other system capable of performing a sequence of logical operations. In one aspect, control system 632 is configured to operate the RAID system 630 nonvolatile memories RAMA 634 and RAMB 636 as a mirrored pair. As such, read/write to nonvolatile memory RAMA 634 are mirrored to nonvolatile RAMB 636, providing redundancy of crucial gaming specific data stored in nonvolatile memory RAMA 634 and RAMB 636. Alternatively, the nonvolatile memory RAMA 634 and nonvolatile memory RAMB 636 may be configured to communicate with control system 632 similar to other RAID storage system levels, such as RAID LEVEL 0, RAID LEVEL 2, RAID LEVEL 3, RAID LEVEL 4, RAID LEVEL 5, RAID LEVEL 6, etc. Further, the RAID system 630 may include more than the two nonvolatile memories RAMA 634 and RAMB 636 shown.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is

calculated to achieve the same purpose may be substituted for the specific embodiments shown. This application is intended to cover any adaptations or variations of the invention. It is intended that this invention be limited only by the claims, and the full scope of equivalents thereof.



Shuffle Master®
GAMING

Shuffle Master
Game Operating System
"SGOS"

Developer's Manual

Rev: May 2001

Overview

Table of Contents

I. SGOS BASICS

II. PROGRAMMING WITH THE API

III. GAME DEVELOPMENT

IV. API TUTORIAL

List of Figures

List of Tables

I. SGOS BASICS

CHAPTER 1 – INTRODUCTION

A. A UNIVERSAL GAME DESIGN APPROACH

Until now the countless video and mechanical games of chance offered for sale have not been at all standardized. Programmers often had to create new code in each new game, for every function, and for each hardware apparatus.

Shuffle Master's new Game Operating System (SGOS) brings a universal game design approach to electronic games of chance. SGOS includes pre-approved software that handles all common game functions and security features without the need for new code.

For simplicity, this manual will refer to the Shuffle Master Game Operating System software as *SGOS*.

B. USES THE STABLE LINUX PLATFORM

SGOS runs on the Linux Operating System. The relatively new Linux platform (conceived in 1991, first introduced more widely in 1994) is fast becoming a favorite for countless embedded applications because it is much more lean and stable than Windows.

SGOS is written in C and C++. You need to have some basic programming skill to create games with the provided templates. You should be proficient in C and C++ if you will be designing new games from scratch.

C. KEY GAME FEATURES

SGOS takes care of all game actions and hardware interfaces. Functions common to all games are pre-approved and located in the SGOS library. Key features of SGOS are as follows:

- All except game personality is pre-designed and pre-approved
- Game initialization and power loss recovery
- Included engines: Draw Poker, Video Reel and Mechanical Reel
- Support for design of new game engines by developer
- Supports both single and multi-game platforms
- Complete animated graphics engine
- Game history and accounting
- Stereo sound support
- Advanced security features
- Drivers for all video game hardware
- Remote debugging
- Online protocol support
- Basic level game development requires only graphic and text changes
- Advanced level game development allows complete design of graphics, sound, and game logic

D. Developing a New Game with SGOS

The SGOS Software Development Kit supports two levels of development:

1. Basic. Use the included game templates (e.g. 9-line) for an immediate design. With a few interesting graphics and minor code changes, a casino tech or other designer can quickly create a unique new casino game.
2. Advanced. Powerful and flexible platform for a totally new game design.

You can quickly design a unique game with one of the provided game templates, design a fully new game with the generic template, or mix the two approaches for efficient best results. However you proceed, your programming time will be slashed with the pre-designed and pre-approved underlying features of SGOS. You can focus on the new aspects of your game design.

E. Potential Users

Any entity that sells, uses, or designs video games of chance can use the Shuffle Master SGOS to develop a unique new game. The most likely users include the following:

1. Game Manufacturers. Manufacturers can save a great deal of design and approval time by using Shuffle Master's SGOS. A manufacturer can create a new game as unique and complex as desired.
2. Casino Operators. With the SGOS platform, a casino can design its own games. Using one of the included game engines and included tools, the casino's programmer can create a unique game with new graphics and a new name.
3. Third Party Developers. A third party programmer or developer may provide a new game design for either a manufacturer or a casino.

F. Target Machines

The examples used in this documentation are based on building a new machine with all new hardware. Refer to *Part VI. HARDWARE SOLUTIONS* in this book for details about commonly supported hardware.

The same game approach will apply for conversion of existing video or mechanical machines. However, the hardware and software interfaces for existing components are very application-specific and are outside the scope of this documentation.

CHAPTER 2 — INSTALLING AND CONFIGURING SGOS

A. PRE-LOADED DEVELOPMENT SYSTEM

If you have a pre-loaded development system from Shuffle Master, you can skip this chapter and start working with SGOS.

B. INSTALL LINUX

If you do not already have it, first install Linux on the computer you will be using for game development. Refer to *Appendix A* for suggestions if you are new to Linux. Red Hat Linux 6.x is preferred, but other Linux packages should be satisfactory.

You can use the most current version of Linux on your development machine, but the build machine must use the Linux version included with SGOS. The Linux OS is a part of the pre-approved code.

Neither Linux nor SGOS is greedy for RAM or hard drive space. You should have at least 32 MB of RAM and 500 MB available hard drive space for a comfortable development environment..

TIP... You can set up your computer as a Windows/Linux dual-boot system. Neither Linux nor SGOS will require very much of your hard drive space. Also, some vendors offer a Linux operating system that runs in Windows. Though it is less efficient, it might be easier to install.

C. INSTALL SGOS

Install the SGOS library and game development files from the SGOS CD-ROM. When you insert the CD-ROM it will lead you through a menu-driven installation.

1. ***Add notes here based on final installation CD.
2. If you want to load the tutorial files now, create a directory named...***add notes

The file tree in *Figure 10-1* shows where the CD-ROM install procedure will place files by default.

File Formats

.ps.Z compressed PostScript file, for UNIX
.tar UNIX archive file
.tar.gz compressed UNIX archive file--"tar fvxz file_name"(to unzip and untar)
.gz UNIX compressed archieve file using gzip--"gunzip file_name" to uncompress
.bz2 Advance compressed UNIX archive file--"bunzip2 file_name" to unzip
.exe Windows executable program
.zip Windows archive file

D. Disable Functions not Supported by Development Platform

SGOS configuration files end with an .oti extension. The file **mygame.oti** provides all settings for the embedded system on a target machine. For your desktop development system, you also must include **local.oti** if you want to disable hardware that is not supported by your development machine. Refer to *Appendix D* for the various .oti file listing and the default version of **local.oti**, which disables the touch screen and other hardware typically not available on a desktop computer running on Linux.

Note that sound is disabled by default, since sound is a bit more complicated to set up on the Linux platform. Modify **local.oti** as needed for your development setup.

TIP... If your mouse does not work, you probably need to change your svgalib settings, as follows:

1. Find **/etc/vga/libvga.config** in the root directory, and open it for editing.
2. Look for your mouse manufacturer and uncomment the appropriate line.
3. If your mouse still is not working, try reversing **mouse_accel_type** to **ON** or **OFF**.
4. You can modify many other mouse settings in this configuration file. Review a Linux manual for further suggestions.

E. Tools Available on the Web

The menu-driven setup process should give you a satisfactory installation. If you are new to the Linux operating system, a Linux manual will help you get up to speed. If you have any problems with the SGOS installation, first look at the README file. Also check the Shuffle Master Web site at **shufflemastersupport.com**.

CHAPTER 3 – SGOS COMPONENTS

A. BASIC SGOS LAYOUT

Although you cannot access the SGOS precompiled library layer directly, it will be helpful to see how it interacts with the game layer and your game program.

1. Library Layer

Figure 3-1 shows the basic layout of SGOS. The bulk of SGOS resides in the library, including the Linux kernel, drivers, event handler, user API, and watchdog. The library tracks all game history and accounting and handles hardware interfaces. The library code is pre-compiled and cannot be directly accessed.

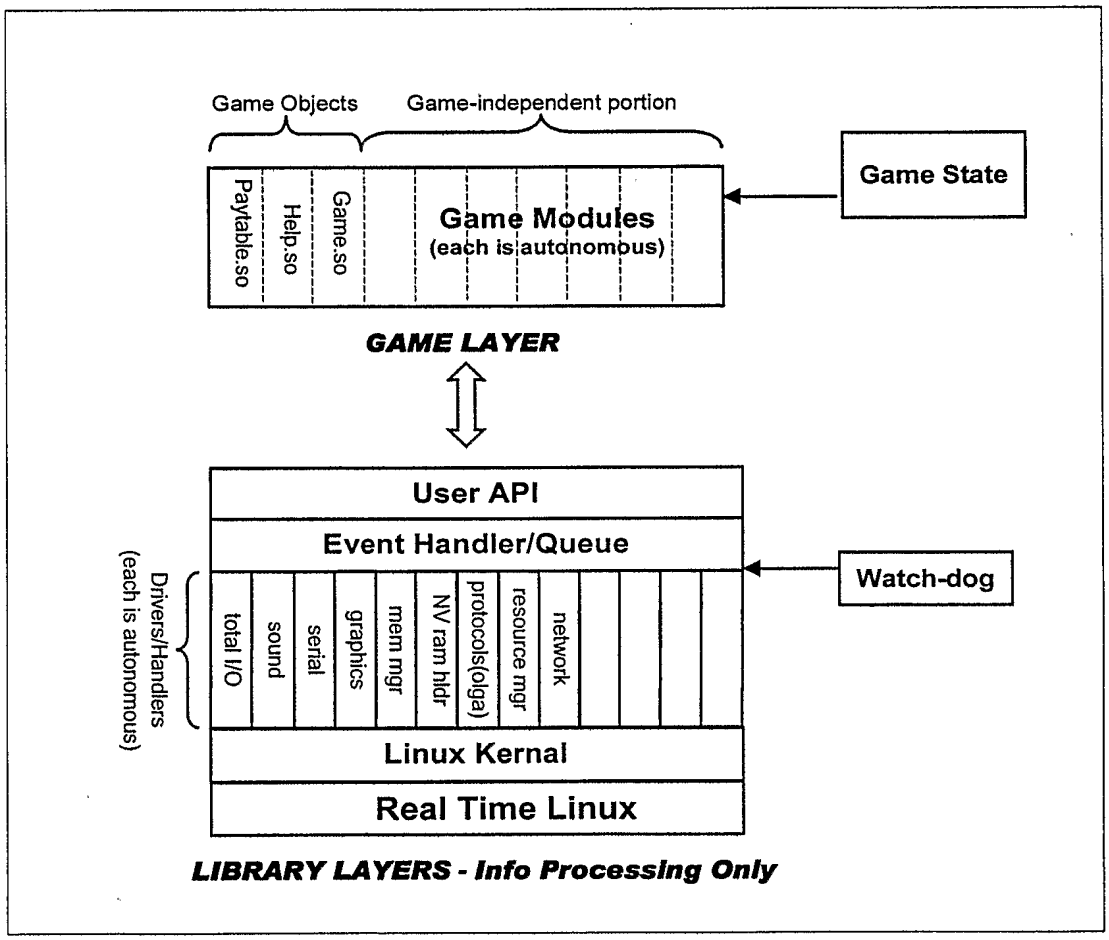


Figure 3-1 – SGOS Architecture Showing Library and Game Layers

2. Game Layer

With the numerous pre-approved features provided in the SGOS library, you will need only a few configuration and object files to develop a new game. The rest of this document will explain how to set up a new game, either by modifying the included game templates or designing a game from scratch.

Version 2.0 of SGOS includes 9-line, draw poker, and generic game templates. *Chapter 10* describes each of the files you will need to create a working game.

B. Linux Real Time and Linux Kernel

Linux 2.2.13 is the operating system for Version 2.0 of SGOS. For consistency with any state game agency approved code, the Linux version of the SGOS library cannot be changed. However, you can develop your game in a newer version of Linux. Refer to *Appendix A* if you are new to the Linux platform.

C. User API

The User API defines all library function calls that can be made from the game layer. The functions called out by the User API provide powerful graphic, sound, and other development options.

Chapter 5 and the tutorials in *Part IV. API TUTORIAL* give an overview and examples of programming the user, engine and game API functions. *Appendix C* provides a complete listing which contains the API functions.

D. Event Handler

Timers and their callback functions are what move a game along in SGOS. The timers are usually tied to graphic events, and may be launched from either of two places:

1. The SGOS game engine (you cannot directly control these callbacks).
2. Your game (you launch timers with the API timer functions).

The event handler queue receives events as they are called, and handles them in the order received. A key feature of SGOS is that events are not threaded together. If a particular event fails, the program will in most cases continue to run. The tutorials show the importance and role of timers, callbacks and the event handler in SGOS programming.

E. nvram and Game State

"Game states" such as **BEGINPLAY**, **PLAY1**, and **EVALUATE** are held in nvram and contain current and historic game data. Game state values provide data to run the current game, display game history, or resume a game where it left off in the event of a power failure or other program disruption. *Table 11-1* lists the primary game states and related callback functions that serve as "hooks" into the game engine.

The pre-approved game engine effectively "runs" the game and handles all game accounting and history. Your game code uses the "hooks" to direct the game engine and create a new game

personality, by defining how the game state callback functions are handled. *Chapter 11* and the tutorials in *Part V. GAME ENGINE TUTORIAL* explain SGOS's important game state feature. It is an innovative and different approach to game design. The examples will help you discover how to work with game states and callback functions in SGOS.

F. WATCHDOG

The watchdog perfoms a hash on every code segment of RAM at regular intervals. The device will tilt if any of these secure hashes change. The watchdog is part of the pre-approved SGOS library.

CHAPTER 4 — TIPS TO GET STARTED WITH SGOS

A. UNIQUE ASPECTS OF SGOS

The SGOS library and user interface are programmed in C and C++. If you are an experienced programmer you probably want to get going with SGOS programming!

Because the pre-approved game engine and library are hidden, you need to know the basics of how they work and how to interface with them. Especially, you will need to understand the following key aspects of the SGOS approach:

- How SGOS uses timers and nvram to run the game.
- How and when to interface with the game engine and game library, especially with the API functions and "game states".

If you skim over the preliminary chapters, keep an eye out for these key concepts.

B. ABOUT THE EXAMPLES AND TUTORIALS

The examples and the tutorial files help to show the unique aspects of SGOS. The files are included on the CD-ROM. You can run them to see simple applications of buttons, timers, nvram, and other SGOS features.

SGOS includes over 100 API functions. The tutorial examples show use of API functions for basic game routines such as buttons and animation. See *Appendix C* for a complete listing of the functions.

The API calls access the SGOS library directly, whereas the callbacks to advance the game states are made via the game engine. To get a quick overview of how these important functions differ and how they are used, refer to *Appendix C* and *Figure 11-1*.

II. PROGRAMMING WITH THE API

Chapter 5 – Scope of userapi Functions
Chapter 6 – Timers, Buttons, and Callbacks
Chapter 7 – Handling Graphics
Chapter 8 – Sounds in SGOS
Chapter 9 – Non-Volatile RAM (nvram)

CHAPTER 5 — SCOPE OF USERAPI FUNCTIONS

A. ROLE OF USERAPI IN SGOS PROGRAMMING

Your game code cannot make standard C calls to the library, including file I/O and the **printf** family. Instead, SGOS provides a large and versatile set of API functions for all the needed routines to create a game. **userapi.h** and **engine_api.h** declare these functions, and you make them available by including **userapi.h** when you build your game.

Like the rest of the library the API is pre-approved and cannot be changed by the programmer. Periodic new SGOS releases will add features. You can also make requests for new API function calls on the support Web site, *shufflemastersupport.com*.

Appendix C provides a full listing of the API functions with explanations.

B. OVERVIEW OF USERAPI FUNCTIONS

1. Graphic Routines

The API includes over fifteen graphics functions to draw and manipulate various shapes. An example of a typical graphics function format and variables passed is

```
gfx_drawbar(int x, int y, int w, int h, int c)
```

which draws a solid rectangle at **x**, **y** with width **w**, height **h**, and color **c**.

Several of the graphics functions, including background, buffer and sprite routines are used for animations. The examples in *IV. API TUTORIAL* provide progressive examples of how to handle graphics in SGOS with these functions.

2. Widget Routines

*** edit this section when we figure out the widget functions

Six widget functions create buttons, set their properties, and enable/disable them. Though all buttons have the same ability to launch an event, SGOS includes three distinct widget types: text only, graphic, and a graphic that changes when the button is pressed.

The "hot spot" function works the same way, except that it does not create any button border, text or graphics. It simply defines an area of the screen that responds like a button if pressed. You can use it for any purpose — for example, to create a button-type response to an area of the screen which a player is intuitively drawn to.

Buttons in SGOS can be disabled and covered with a new button, but cannot be modified or removed until the current module is closed. You may want to use the hot spot along with defined rectangles to build more complex button schemes which can handle changes.

3. Module Handling Routines

Three module routines load, jump and exit a module, and either save the current module

name for retrieval or revert to the previous remembered module.

4. Timer Routines

The two timer routines — `timer_start` and `timer_kill` — are very important in SGOS event driven programming. For more information refer to *Chapter 6* and the tutorial examples in *Chapter 20*, *Chapter 19*, and *Chapter 21*.

5. Non-volatile RAM Routines

The nvram routines manage stored **`mygame.state`** values on the nvram hardware (or in the nvram file on the development platform). The game application cannot directly touch nvram. All game manipulations and retrievals of nvram occur through API functions. Refer to *Chapter 9* for an explanation of how the SGOS nvram manager works.

The eighteen API functions that work with nvram perform any of the following three actions on the six different types of strings (char, short, int, long, float, double):

1. Set a value
2. Retrieve a value
3. Increment a value

6. Sound Routines

API sound routines are available to play or stop .wav sound files and set the master volume level.

7. Mechanical Reel Routines

Specific routines targeted for use with mechanical reel slot machines. Refer to *Appendix C.*

8. External Display Routines

Specific routines targeted for use with external displays. Refer to *Appendix C.*

9. Text Formatting Routines

Three text formatting functions are available for formatting text and numbers.

10. Resource Routines

11. System Routines

Miscellaneous API routines include a random number generating routing, setting a lamp defined within the .oti file, and debugging routine. For setting up the debug and for setting a breakpoint refer to *Chapter 14-E, Debugging Tools.* Two revision information functions return strings with the version and release date of the running version of SGOS. Note that the user's game version is set in the file "version" in **`app.temp`**. This file is created/updated at compile time.

12. Engine NVRAM Routines

These API calls let you check values in the **engine.state** and other parameters set by the game engine.

13. Multigame Management Routines

14. Miscellaneous Game Engine Routines

15. Game Specific Routines

Refer to *Appendix C* for API calls that are specific to nineline or draw poker games.

CHAPTER 6 — TIMERS, BUTTONS, AND CALLBACKS

A. EVENT-DRIVEN PROGRAMMING

A game in SGOS is event-driven. Timers and their callback functions are the motive force in SGOS that cause a game to proceed and move along. The timers are usually tied to graphic events or player events such as a coin drop or button press. A timer can launch from either of two places:

1. The SGOS game engine (you cannot directly control these callbacks).
2. Your game (you launch timers with the API timer functions).

Even though you can react to or create events from the game side, every event is actually serviced on the library side of SGOS.

B. LAUNCHING TIMERS

In an SGOS game, you launch all timers and timer callbacks using the API timer routine:

```
timer_start(long timeout, char* callback)
```

This function passes the timeout period in milliseconds (1000 milliseconds = 1 second) and a callback function name (not a pointer). Once you start a timer, it times out, then sends the event (the callback function) to the event handler queue, which handles all timer events in the order they are received.

Note that **char* callback** is not called with any parameters. You must declare timer callbacks as **void** functions.

C. MULTIPLE AND PERIODIC TIMERS

By default each timer is good for only one call. You can start multiple timers either by calling each one separately, or by setting up a periodic timer.

1. Starting Multiple Timers

You can start as many timers as you need, each with a separate call to **timer_start()**. It is acceptable to call the same callback multiple times; the event handler will track each occurrence. For instance,

```
timer_start(100,"draw_screen");
timer_start(200,"draw_screen");
```

will call **draw_screen()** in 0.1 and 0.2 seconds from now.

SGOS has a system limit of 60 timers at any one time. You should never need this many active timers, but the limit is in place because more than 60 timers can create a bottleneck on a typical processor for a target machine.

2. Setting Up Periodic Timers

Create a loop where the callback function's last step re-initiates the timer to call itself back. This manual will refer to a timer that calls itself back as a "periodic timer." *Chapters 20 and 21* show tutorial examples that use periodic timers to launch several animated balls that move around the screen.

NOTE... Events in SGOS are not threaded together; if a particular event fails, the program will in most cases continue to run. With periodic timers (with callback functions that call themselves back) you can start animations once and not have to worry about them again.

Chapters 19, *20,* and *21* show examples of how to use API timer routines.

D. Using Timers for Screen Updates

SGOS event-driven programming makes extensive use of timers. With every event on its own timer you can easily add and remove items from the screen.

Timers are not associated to numbers, so it can be a puzzle to track a particular timer through the **debug.out** file. Looking in the **debug.out** file is more useful to see how many timers are executing at one time.

E. Using Timers for Animation

You can create single or periodic timer events. A single timer event occurs once, after a specified number of milliseconds. A periodic timer event occurs every time a specified number of milliseconds elapses. The interval between periodic events is called an event delay. Periodic timer events with an event delay of 10 milliseconds or less consume a significant portion of CPU resources.

The relationship between the resolution of a timer event and the length of the event delay is important in timer events. For example, if you specify a resolution of 5 and an event delay of 100, the timer services notify the callback function after an interval ranging from 95 to 105 milliseconds. A mix of logic and experimentation are your best tools to find the right ranges for your game design.

F. Killing Timers

You can cancel an active timer event at any time by using the `timer_kill(char* callback)` function.

WARNING! Be sure to pass in the timer name. If you pass a null, the function will default to killing all timers, including all operating system timers. This will cause the system to crash.

Note that the multimedia timer runs in its own thread.

G. BUTTON EVENTS

The SGOS API offers three button types and a "hot spot." (Refer to *Appendix 3* and *Chapter 19* for details and examples of button behavior). The buttons and hot spot each look different on the screen, but they all launch an event in the same fashion.They define a screen area that triggers a callback when touched on the touch screen (or with a mouse click on your development computer).

For example,

```
makebutton1(char* name, int x, int y, int w, int h, int basecolor, char*
callback)
```

defines a simple button with the text string **char* name**. When pressed, the button calls **char* callback**, and the callback function is placed in the event queue. It works the same as the timer callbacks discussed above, but without a timeout period. The button serves as another event in the event-driven SGOS scheme.

CHAPTER 7 - HANDLING GRAPHICS

A. A DIFFERENT APPROACH TO GRAPHICS

You will likely find the SGOS approach to graphics and animation to be different from other approaches you have used. At first the event-driven programming, timer callbacks, and graphics routines may seem indirect compared to typical C programming. But as you spend time with it you will probably come to see it as a very powerful, flexible, efficient, and even "forgiving" programming environment.

B. XPM GRAPHIC FORMAT

The XPM graphic format is a pixel-by-pixel array of an icon's colors. You must convert all your game graphics to XPM format to work with SGOS API routines.

You can compile the converted XPM graphics directly into your program or else have SGOS load them from files. Compiling them into the program improves performance, especially on the embedded system. This manual refers to the graphic items you include in your program as "icons."

- Transparency images need to be set to RGB value (255,0,255).
- Save the image with transparencies in a file format such as TIFF (which does not dither the image). This way the non-transparent image will not have a halo around it. The halo is caused when the file format such as JPG dithers the image edges. Any color other than RGB (255,0,255) will not be transparent.
- All animations need to be saved as separate frames. Later enhancements may include the ability to play AVI or MPEG files.
- To conserve memory, try and make all animations with as few frames as possible. For example, use one image to show any static picture instead of multiple images at rest. Make all frames of animation the smallest size to show all the animation sequence. The bigger the image, the more memory it uses. In other words, why use the entire screen with transparent color background when the animation takes place in only a fraction of the total screen area.

Several of the API routines support transparency, as discussed in *Sections G* and *H* of this chapter.

C. CONVERTING ICONS TO XPM

SGOS provides a Python script tool, **convgfx.py**, to convert your graphic icons to XPM arrays. The script converts your graphic image files, organizes them into named XPM arrays in a new C file, and creates a related header file.

The basic approach to use **convgfx.py** is as follows:

1. Put all your graphics files in one directory (typically **app.temp**).
2. Each icon filename becomes its XPM array name. For example, either **icon1.tif** or **icon1.jpg** will convert to an XPM array named **icon1** (the file extension is dropped).
3. When you run **convgfx.py**, it will stuff all the icons into a new .cpp file and create a

header file. For instance, typing

```
[app.temp] # ./convgfx.py mygame_icons
```

at the prompt creates **mygame_icons.cpp** and **mygame_icons.h**.

Chapter 17 shows the use of **convgfx.py** in a tutorial example. *Appendix G* lists the **convgfx.py** file. You may want to modify it to fit your particular needs.

D. Organizing Icon XPM's

After you create the .cpp file containing all your XPM icons, you can take the further step of organizing logical icon groups into arrays. You will probably want to collect the icons for each animation in your game into an array. For instance, an array named **animation_1[]** might include **icon1**, **icon2**, **icon3**, etc.

You can spread your icons among several files, or even maintain a separate file for every icon However, it is usually simplest to include all of your graphic icons in a single file. *File Listing 7-1* shows an excerpt of the first few lines from a file that lists all XPM-formatted icons for a nineline game.

Note that this file (*File Listing 7-1*) has been rearranged into logical arrays for housekeeping. The example includes one array for all of the bitmap reel icons, plus other arrays and single icons as required for the game (not all are shown in the excerpt). The first array, starting on line 6, names all of the icons for the reel strips. The second array, starting on line 19, lists the icons for a very simple animation that happens to include one of the reel icons.

The file also contains named icons that are not members of arrays. The XPM listing of the first icon, named **bell**, begins on line 25. Note in lines 26 and 27 that the numbers inside the double quotes define a 120 x 120 pixel array using 255 colors given as two-character ID's.

The pixel-by-pixel array appears immediately after the colors, in this case using two-character ID's. Line 29 shows the beginning of the lengthy pixel-by-pixel color listing. SGOS also supports single character ID's for files with fewer colors. See *File Listing 17-1* for an example of an XPM listing for a simple ball icon.

File Listing 7-1: nineline_gfx.cpp, Example of xpm Listing

```
#include "userapi.h"
/* XPM */
#include "nineline_gfx.h"

// An array containing all the icon bitmaps
const char* const * icon_bitmaps[] = {
   whambutn01,
   cherries,
   oranges,
   melon,
   plum,
   cruise02,
   ring02,
   car02,
   bigbucks01,
   pyllogo01,
   pwham201
{;
const char* const * bigbucks_anim[] = {
   bigbucks01,
   bigbucks02,
   bigbucks01,
   bigbucks02
};
const char* const bell [] = {
/* width height num_colors chars_per_pixel */
   "   120   120    255            2",
/*colors*/
  "..c #54214",
  ".#c #baad09",

  .......etc rest of the colors
```

See File Listing 17-2 for a more complete example of a simple XPM graphic.

E. Three Graphics Buffers: Screen, Background and Frame

The following three buffers are available to manipulate graphics in SGOS:

1. **GFX_SCREENBUFFER** – Whatever is drawn to the **GFX_SCREENBUFFER** will display on the screen.
2. **GFX_BACKGROUNDBUFFER** – The **GFX_BACKGROUNDBUFFER** is generally for storing the complete original **GFX_SCREENBUFFER**. You can use various API routines to restore the **GFX_SCREENBUFFER** when you remove or relocate icons.
3. **GFX_WORKBUFFER** – This buffer is optional. The **GFX_WORKBUFFER** provides more options for how you assemble icons before drawing them to the screen buffer, especially for more complex animations.

F. Setting the Buffer Context for API Functions

The API graphics routines can interact with any of three buffers. To use these routines you

must first set the proper buffer, or "context", with the API call

```
gfx_setcontext(int context)
```

where **context** is one of the three buffers: **GFX_SCREENBUFFER**, **GFX_BACKGROUNDBUFFER**, or **GFX_WORKBUFFER**. The SGOS library assigns integer values to control the outcome of all API routines that need to know the context.

TIP... You do not need to reset the context if you can logically determine that the most recent setting is the one you need. If quite a few lines of code have occurred since the last context was set, you may want to set it again to protect against future inserted code.

G. Colors Reserved for Transparency

SGOS assembles a transparency by replacing a graphic's transparent pixels with pixels from the active screen or the buffer (depending on the function used). You must use the right functions and buffer contexts in the right order to get appropriate transparency results.

SGOS enlists a transparency buffer whenever you use an API routine that includes transparency. For color schemes that can be covered by single hexidecimal characters (i.e. 45 or less) the graphics conversion tool will assign the character **9** to transparent pixels. For larger color schemes it will assign the two characters **99**. Both of these equal pure magenta.

WARNING! Pure magenta is not a common color in most artwork. If it does appear in a graphic that also includes transparency, you will get improper results — all of the pure magenta pixels will show up as transparent since they are defined by the same characters (**9** or **99**). You will have to shift the pure magenta to a slightly different color in your original graphic to keep it from being treated as a transparency.

H. "Trans" and "Sprite" Transparency Functions

Both the "Trans" and "Sprite" types of API transparency routines create transparency by substituting pixels from a buffer. The two types of functions can sometimes accomplish the same task, but they differ considerably in how they process data.

Key aspects of the Trans routines are as follows:

- They let you define a transparency color (can be any color).
- They get transparency pixels only from the **BACKGROUNDBUFFER**.
- They are drawn to a special interim buffer.*
- They are not cached; they are slower, especially on a target machine.
- They are best for static screen updates or very low level animation.

A notable Trans exception is that by calling **gfx_drawpartXPM() loaded with all -1 arguments you can cache the entire graphic, as a sort of pseudo-buffer.*

Key aspects of the Sprite routines are as follows:

- They always use a transparency color of **99** or **9**.
- They can get their transparency pixels from any of the three buffers.
- They offer a fill instead of transparency with **setspritetype(SPRITEFILL)**.

- They are cached and have no interim buffer; they are much faster.
- They are the tool of choice for complex animations.

The discussion of animation approaches in this chapter and the tutorials in *Chapters 19, 20,* and *21* will help you see the applications of the various transparency functions in SGOS.

I. Drawing to the Three Buffers

All three graphics buffers in SGOS use an area of memory with the same number of bytes as the screen. Multiple API functions can draw strings, lines, bars or graphic items to any of the three buffers – **GFX_SCREENBUFFER**, **GFX_BACKGROUNDBUFFER** or **GFX_WORKBUFFER**, whichever is set as the current context. Many of these functons use the syntax **gfx_draw***[name of action]*(). Some of the draw functions flip or otherwise manipulate a drawn item, and/or draw it as a transparency. The API button calls (e.g. **makebutton1()**) should only be made to the screen (context set to **GFX_SCREENBUFFER**), to ensure that proper functionality is available on the screen. Once a button is created, you can update the button's graphic properties (e.g. **setbuttoncolor()**) in either context. Make sure that you modify button properties on the background buffer if you are going to copy the background to the screen.

See the tutorial in *Chapter 19* and the API calls in *Appendix C* for more about handling the graphics aspects of buttons.

J. Updating Among Buffers with GFX_COPYBUFFER()

The API function to copy to or from the different buffers, is

```
gfx_copybuffer() - copies all or part of one buffer to another
```

You will be using this function extensively as you make animations in your game.

K. Limited Animation Using Only the SCREENBUFFER

The simplest animation to set up in SGOS is a series of rectangular icons without transparency that remain in a fixed area of the screen. In this simple case, you can draw each new icon and simply let it replace the prior one. Neither the **GFX_BACKGROUNDBUFFER** nor **GFX_WORKBUFFER** would be needed.

However, you will need at least the **GFX_BACKGROUNDBUFFER** if you have any of the following very common animation criteria:

- Transparency
- Change in icon size or position
- Overlap with any other updated icons

Without another buffer in these cases, you might get lingering fragments from prior icons, absent background areas, or unwanted transparency fills.

L. Using the BACKGROUNDBUFFER for Transparency

You can store an exact copy of your **GFX_SCREENBUFFER** to the **GFX_BACKGROUNDBUFFER** at any

point with **gfx_copybuffer()**. The "Trans" functions use the current data in your **GFX_BACKGROUNDBUFFER** to fill in transparent pixels. (The "Sprite" functions can use any of the three buffers, whichever is set as the current context.) The tutorial examples in *Chapters 17 and 18* show examples of how transparency works in SGOS.

You will generally want to create your initial screen and copy it to the **GFX_BACKGROUNDBUFFER** before you begin any screen changes or animations. All buttons must be created to the **GFX_SCREENBUFFER** so they can launch appropriate functions with the mouse or touch screen. (Also refer to the discussion of buttons in *Chapter 6* and the tutorial example in *Chapter 19*.

M. Double-Buffered Animation

Besides being the most likely source of transparency pixels, the **GFX_BACKGROUNDBUFFER** is necessary for any screen change or animation that requires graphics data from the original screen. For "double-buffered" animations, the **GFX_BACKGROUNDBUFFER** is the repository for restoring pixels when you move any icon to expose an area that must be restored to the original screen content.

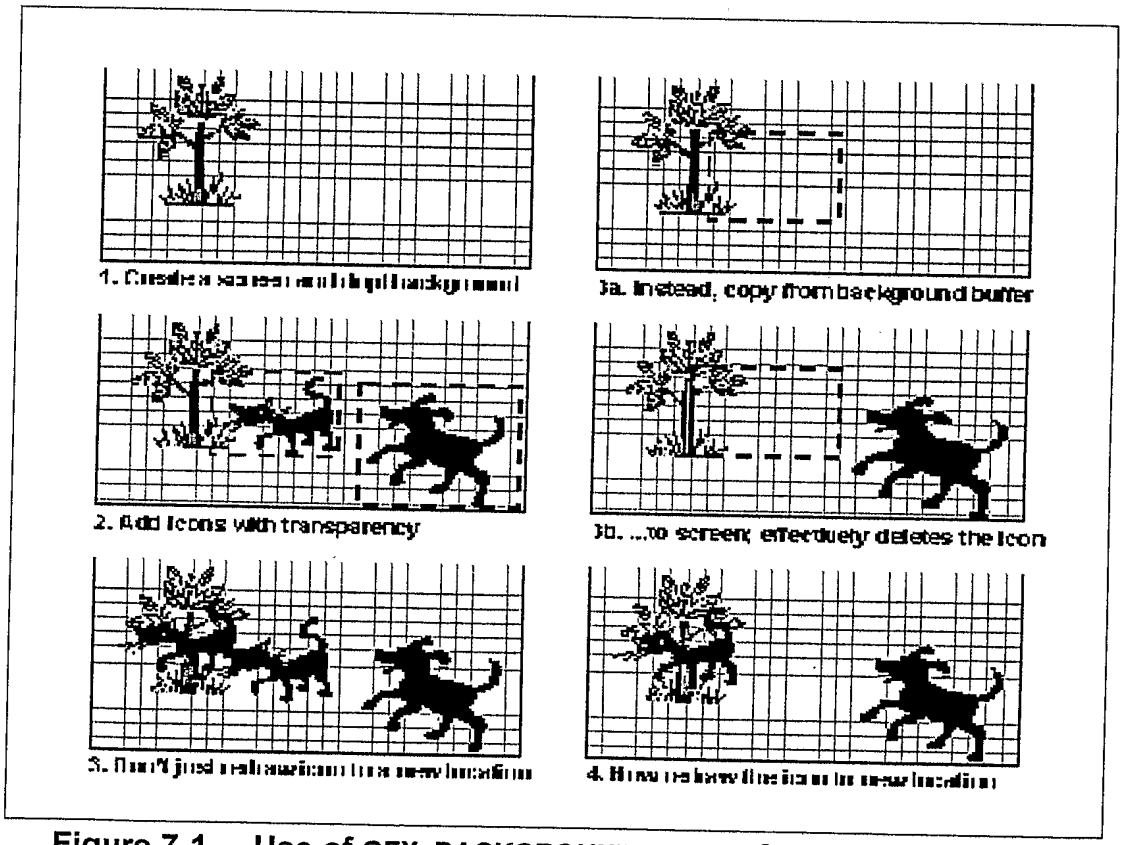


Figure 7-1 – Use of GFX_BACKGROUNDBUFFER for a Screen Update

Figure 7-1 shows an elementary way to use the background buffer in an animation, as follows:

1. Create a screen using API functions as needed, then create a duplicate background buffer using

`gfx_copybuffer(x, y, w, h, destx, desty, GFX_SCREENBUFFER, GFX_BACKGROUNDBUFFER)`.

2. Add one or more icons using any of the following functions which support transparency:
   - "Trans" functions — `gfx_drawtransXPM()`, `gfx_drawtransFile()`, or `gfx_rotatetransXPM()`
   - "Sprite" functions — `gfx_drawsprite()`, `gfx_drawspriteflipped()`, `gfx_drawspritepart()`, or `gfx_drawspritepartflipped()`. First set the graphics context to the `GFX_BACKGROUNDBUFFER` and the sprite type to `SPRITETRANS` with `gfx_setgraphicscontext()` and `gfx_setspritetype()`. (Note that the graphics context has no effect on the "Trans" functions, because the context is hard-wired for these functions.)
3. Wrong way and right way: Step 3 in Figure 7-1 shows what happens if you merely redraw the icon to a new location. The old icon remains, and the new one is placed next to it.

   3a and 3b show the right way to erase the old icon and restore that area of the screen. Use the API call

   ```
   gfx_copybufferpart([source (x,y,w,h)],[destination (x,y)],
   GFX_BACKGROUNDBUFFER, GFX_SCREENBUFFER)
   ```

   to retrieve the needed rectangle. Depending on how you first drew the icon, you may need to call `gfx_getdimensionsXPM()` to find `w` and `h`.
4. Now you can successfully draw your new icon to the new location (this simple example draws the same icon to a new location), using an appropriate trans or sprite API call.

N. Role of Timers and Callbacks in Animations

SGOS event-driven programming relies heavily upon timers and callback functions to carry out animations. With every event on its own timer you can easily add and remove items from the screen. Also, by using functions which call themselves back via timers, you can start animations and let them run. You can let another event modify the animation or kill the timers.

Refer to *Chapter 6* and the tutorial examples in *Chapters 20 and 21* for more about animation using timers, callbacks, and the event queue.

O. Uses for the GFX_WORKBUFFER

You can accomplish virtually any animation using just the two buffers. However, having the `GFX_WORKBUFFER` as a third provides many more options for assembling complex graphics and animations.

For example, if you use the animation technique commonly called "dirty rectangles", you will need a third buffer. "Dirty rectangles" tracks the smallest rectangles of the screen that truly need to be restored, which can drastically reduce the need for memory resources. For instance if an icon only moves a few pixels, very little of the screen would need to be restored because it remains covered by the new icon.

CHAPTER 8 — SOUNDS IN SGOS

A. WAV FILES

SGOS offers powerful sound capabilities that are simple to use.

As you are designing your game, format all of your sounds as wav files and place them in the app.temp directory. You will manage all of your game sounds with a few API calls. Playing Sounds

You play the sound file named char* file using the API function

```
sound_play(char* file, int channel, int loop, int pan).
```

You can set whether the file loops, pick any of 32 channels, and fade left-right (2 to -2). For example,

```
sound_play(mysound, 15, 1, -2)
```

would set the channel to 15, cause the sound to loop, and pan fully to the left speaker. The sound would not stop until you call **sound_stop(15)**.

NOTE... Changing modules (**mod_exit, mod_load**) does not stop the sound play. For example, if there is a looping sound playing during the main game play and the bonus is hit, when the bonus module is loaded the recurring main play sound will need to be manually halted with the **sound_stop** command.

Changing modules does not stop the sound, i.e., leaving the main play screen with sound playing and entering the bonus module would still have the main play sound playing.

You set the master sound volume with **sound_volume(int percent)**, as a percentage of the hardware setting.

Refer to the complete API lisings in Appendix C for a more complete description of all the sound API calls.

CHAPTER 9 – NON-VOLATILE RAM (NVRAM)

A. NVRAM ROLE IN SGOS

NOTE... Your development computer will use a file in the **app.temp** directory called **nvram** to represent the target machine's non-volatile RAM. For simplicity, this manual refers to both the non-volatile RAM hardware and your development **nvram** file as "nvram". Most references will be to the **nvram** file, since the manual presents game development concepts. *Chapter 24* discusses Shuffle Master's nvram hardware solution for the target machine.

With SGOS you will use flash memory or other type of non-volatile RAM (nvram) to preserve needed game data across executions. SGOS stores two basic types of data in nvram:

1. Data you set for your game. You can set up structures and variables as needed to store data needed across executions among your game modules. You cannot directly access the nvram, and must use API functions to set, get, and modify your game's nvram values.
2. Data the library maintains. The nvram stores game history to provide accounting records, and tracks "game states" such as **INITIALIZE**, **PLAY1**, and **EVALUATE** to tell the game engine what it is supposed to be doing and where to resume in case of a power failure.

This chapter covers only the first kind of nvram data and the API functions you will use to store data in one module and retrieve it in another. *Chapter 11* will cover the crucial game state concept and how you work with it in designing a game.

B. SETUP OF NVRAM DATA WITH MYGAME.STATE

Your game application cannot directly touch nvram. The SGOS library includes a special set of routines to manage the nvram data as a text file. All game manipulations of the nvram occur through these functions. The **mygame.state** file defines how your game level nvram values will be stored.

You will want to give careful thought to organizing the data you want to store in nvram. The **.state** file makes it easy to put variables in the nvram, since the game application never needs to be concerned with the offsets of nvram variables. Structure definitions can even be nested.

You use an **nvram string** to refer to a specific variable in nvram, for example **Game.creditsleft**. You can access arrays with standard **printf**-style format characters, with values inserted as the additional arguments on each line. For instance you could write strings such as

```
History[%l].bet,
Stats.icon[%l],
```

or even

```
BillHistory[%l].date[%l].
```

More elaborate formats are also possible, such as **MyStruct.%s[%l]**, since the string is format-

ted before being parsed.

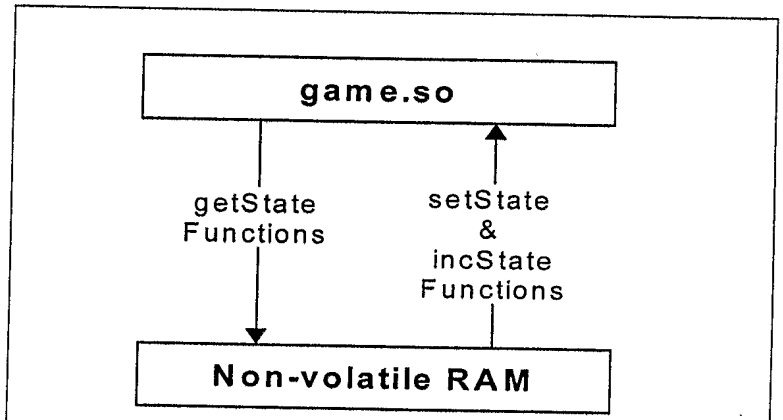
The nvram handler partitions nvram into variables, guided by the contents of the **mygame.state** file. This text file looks similar to a list of C structure declarations, except that it uses # as the comment character to show lines that will not be parsed. For example, the tutorial in *Chapter 20* uses the following code to set up a structure **ball** that will hold icon location and velocity data for up to fifty animated balls:

```
# Optional comment line
struct ball{
    int x;
    int y;
    int x_vel;
    int y_vel;
    int handler;
    } Ball[50];
```

The library's nvram handler reads and parses this data at startup. The nvram structures are frozen for a completed and compiled game design. Your game.state file defines the variables on the target machine before your program ever runs. Even on your development computer, you cannot add new variables or change the size of an array while your program is running.

C. NVRAM API FUNCTIONS

Figure 9-1 shows schematically how the nvram API functions communicate between your



Notes:
1. *See userapi appendix for complete nv_set & nv_get functions.*
2. *A development computer simulates nvram with a file named nvram in the directory user/local/sgos/shared.shfl.*

Figure 9-1 – Use of nvram with game.so

game and nvram.

You input or update nvram variables with the **set_state**, **get_state**, and **inc_state** functions. The API includes six of each of these functions, one for each variable type: **Char**, **Short**, **Int**, **Long**, **Float**, and **Double**. For example, the tutorial in *Chapter 20* uses the following function

calls to set and retrieve a variable named **ball.x**:

```
nv_setint("Ball[%d].x",balls[i].x,i), and
nv_getint("Ball[%d].x",&balls[i].x,i)
```

Refer to *Chapter 20* to see the complete example, including the **mygame.state** file and example game code. *Appendix C* explains the arguments for each function.

D. CLEARING NV-RAM

On your development computer, you must clear the nvram to reset it whenever you add or remove variables or change the layout of the nvram. To clear the nv-ram on the development platform simply delete the nvram file. SGOS will detect the nvram clear and then reinitialize it to zeros.

E. NVRAM CALLBACKS

SGOS provides for "nvram callbacks". Each callback function is associated with an nvram variable. When that varable's value is changed, the callback is called. ***Does this just refer to the game hooks, or can callbacks be passed with the API functions?*** Elaborate.

III. GAME DEVELOPMENT

CHAPTER 10 – FILE STRUCTURE

A. FILE TREE

Figure 10-1 – SGOS File Tree shows most of the SGOS files you will see and work with.]

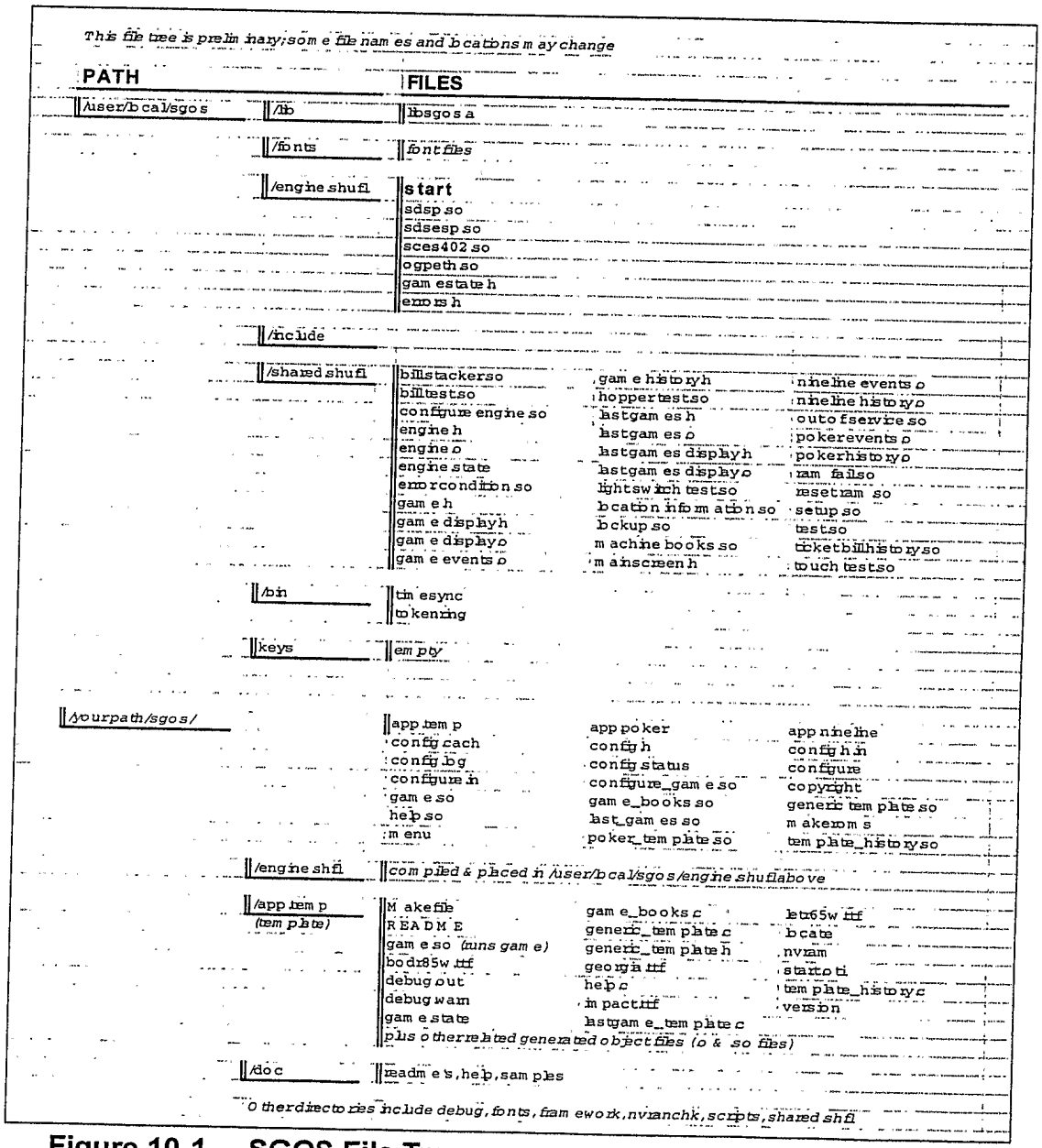


Figure 10-1 – SGOS File Tree

B. Required Files

Table 10-1 describes the key files in your app.temp game directory.

Table 10-1. Developer Game Files in app.temp Directory

File Name ! = Make utility generates file		Description	Details; Chapter & Appendix refs
debug.out	!	Generated	Text file of debug results; Chapter __
debug.warn	!	Generated	Text file of nvram variable type debug errors, Chapter __
fontnames.ttf		Font files	Include all font files needed for game
game.state		Nvram – game specific data	
game_books.c		Screen	Game books screen "winning statistics"; developer provides data
generic_template.c		Game template	
generic_template.h		Game template	
help.c		Help screen text, layout	
lastgame_template.c		Game history screens	Game history screens & functions; scroll to last and previous games
local.oti		Hardware settings	Optional file - include to disable coin hopper etc not set up with devel computer
Locate		Script	Used by Makefile to find engine.shufl & shared.shfl directories
Makefile		Generates game files	See Appdx __. Makes files to user/local/sgos/shared.shfl
nvram		Generated by game	Holds nvram data for current + 69 previous games; includes certain engine.state data plus user's game.state data
Readme		Guidelines, updates	Appendix __ ; see CDROM for current file
start.oti		Resource initialization	Chapter __ and Appendix __
template_history.c		Saves history	Contains functions to save game history
Version		User's version no.	User assigns version no. to track game code updates
.o and .so files		Compiled object files	Placed in user/local/sgos/shared.shfl; refer to Filetree, Fig __

CHAPTER 11 – GAME STATES AND MANAGING THE GAME ENGINE

A. HOW THE SGOS GAME ENGINE RUNS YOUR GAME

The SGOS game engine and library run all routine game functions and drive the gaming hardware. As far as possible, code that could be made common to all games resides in the precompiled SGOS library. Creating a functional game in SGOS has two primary aspects:

1. Develop the game personality using API functions and C programming.
2. Use game engine hooks to "steer" the actions of the game engine and library.

This chapter will show you how the game engine drives a game, and how you steer where it's going. This steering of the game engine may seem a bit indirect at first. The game engine runs the entire game and only offers you a handful of hooks to guide it. The beauty of this approach is that all of your housekeeping and accounting and hardware interface is already done. You tell the engine at a few key points how it should incorporate your game personality into its grand scheme of safely and securely running the game.

B. INTERFACE BETWEEN GAME AND LIBRARY LAYERS

There are three primary mechanisms that the game and library layers use to pass instructions, executions, and results back and forth, as shown in Figure 2. These are the User API, timer mechanism, and callback mechanism.

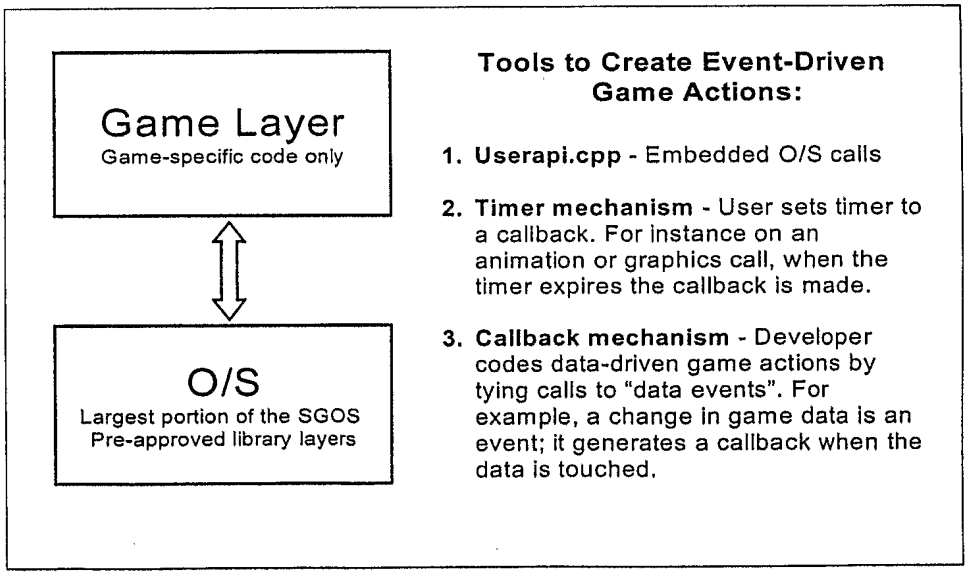


Figure 11-1 – Event-Driven Game Actions

C. EVENT QUEUE

The event queue is at the center of the SGOS event-driven game approach. Typical events are covered later in this chapter.

1. Startup and Idle Mode

SGOS starts up as follows:

Boot Loader — The boot loader loads everything it is instructed to load, then looks for start.cpp.

Start — Start is the "main" executable. It launches the program as follows:

Initializes operating system tasks

Parses the **.oti** file and starts Resource Manager

Parses **game.state** file (NV RAM handler)

Loads **game.so** (game module)

Game.so — Begins execution of the game engine logic.

Idle Mode — Game is now ready and waiting for an "event" to move the game along.

2. Events

SGOS is fully event driven. When an event occurs, such as a coin or bill being entered, a program action or series of actions are set in motion. When the event — the program action or series of actions — is completed, the game returns to its idle mode and awaits a new event. The callback functions and timers which drive the events are discussed below.

The game developer can react to or create events from the game side, but all events are serviced on the library side of SGOS.

3. User API

The game layer and deepest library layers of SGOS cannot directly communicate. These two layers interact via the userapi, data callbacks, and timer callbacks. Data callbacks and timer callbacks will be discussed below.

4. Callback Functions as Game Hooks

The game engine offers callback functions as "hooks" into the game. These callbacks let the game developer arrange the needed game events into a unique new game design. All graphics, animation, sounds, and rules and flow of the game can be manipulated with callbacks in the developer's game engine. For instance,

Insert example from template.c

defines a base callback into the game layer. The callback function gives the game developer a "placeholder" to readily add game-specific logic to the base game engine.

D. TIMER CALLBACKS

Timers are a special API function to place events in the event queue. Both animation and sound tasks are carried out with timer callbacks. The timer format is

```
start timer(long timeout, char* callback)
```

The timeout period in milliseconds is the time until a callback is placed on the queue. Once on the queue it is handled on a first-come, first-served basis. Sequence of timer actions should be coordinated to ensure that a needed action is not placed too far down in the queue. Avoid timers for mission critical tasks.

E. GAME STATES AND NVRAM

As noted above, SGOS stores the "state" of various game aspects after any occurrence of events. An event can be any of the following:

- an action by a player of the game,
- a subsequent action if the player action causes a chain of events, or
- an error.

[Further discussion figures, table, and refer to tutorials]

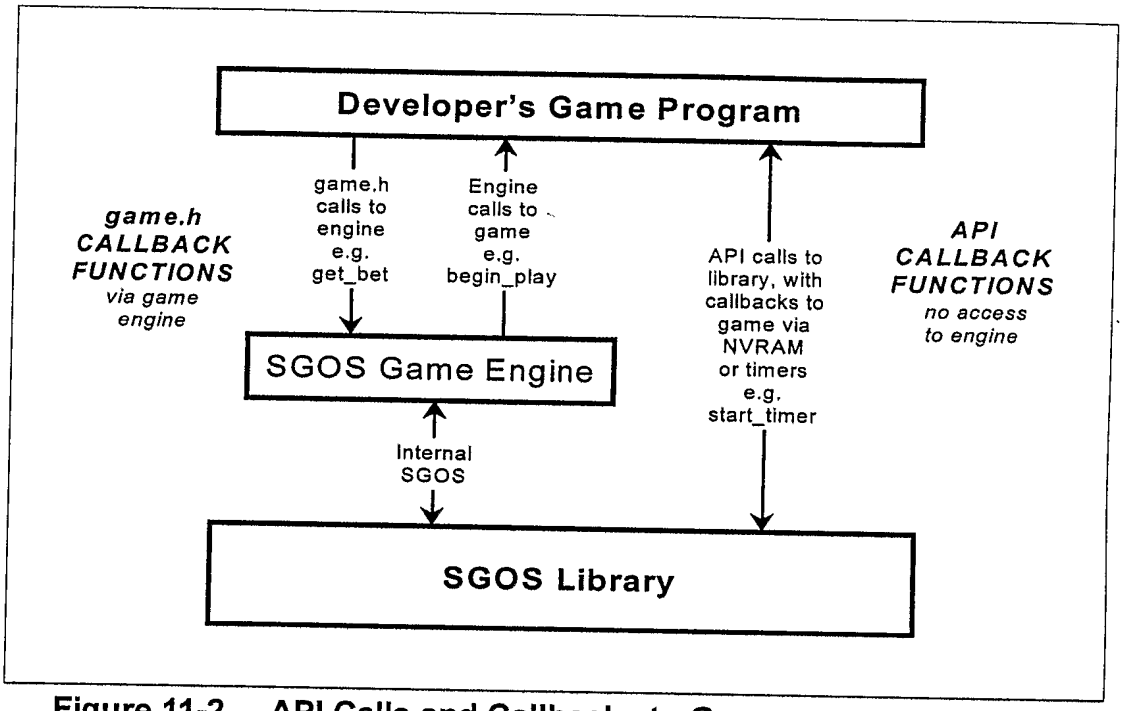


Figure 11-2 – API Calls and Callbacks to Game

Table 1 shows the five key game states that drive a typical game, and lists the primary callback

Table 11-1. SGOS Primary Game States and Callback Functions

PRIMARY GAME STATES	GAME ENGINE CALLBACK FUNCTIONS	NEXT STATE SET BY
Game is always be in one of these basic states.	*Hooks from the game into the game engine.*	*Most set by game engine*

Table 11-1. SGOS Primary Game States and Callback Functions

Initialization Start up or reset game	init_game reset_game cache_gfx draw_game_screen enable_max_bet disable_max_bet	SGOS Engine
IDLE Game up and ready for play	attract animate_idle	SGOS Engine
BEGINPLAY Initiate with coins or other credits	begin_play maxbet_game pick_numbers update_lights update_buttons	SGOS Engine
PLAY1 First play sequence	play_one update_lights update_buttons	Game
PLAY2 Subsequent play sequence	play_two update_lights update_buttons	Game
EVALUATE Evaluate for win or bonus **4A. BONUS** **4B. JACKPOT**	evaluate_game check_for_jackpot check_for_bonus bonus animate_winner award_sound update_lights update_buttons	SGOS Engine
FINISH Payouts and bookkeeping	finish_game update_lights update_buttons	SGOS Engine

Note: The above callback functions are hooks into the SGOS game engine and library. The game engine performs housekeeping and hardware functions, and provides the callbacks as appropriate to move the game along. The game developer creates a new game personality by defining how the callback functions are handled.

functions that must occur for the game to run. Numerous other callback functions are available for game design, but the game engine looks for these primary callback functions to advance through a game.

The engine includes twenty-two states, many of which you will not see at the game layer. The following is a complete list of SGOS game states, as defined in **gamestate.h**:

IDLE
BEGINPLAY
PLAY1
PLAY2
EVALUATE
FINISH
CREDITSIN *(no game-specific aspects)*
COUNTDONN *(no game-specific aspects)*
JACKPOT *(no game-specific aspects)*
HANDPAY *(no game-specific aspects)*
SETUP *(no game-specific aspects)*
PAYOUT *(no game-specific aspects)*
BETTING *(no game-specific aspects)*
BONUS
BONUSCOUNT *(no game-specific aspects)*
MAINMENU *(no game-specific aspects)*
AUTHORIZE *(no game-specific aspects)*
COUNTDOWNHOPPER *(no game-specific aspects)*
TEST *(no game-specific aspects)*
HOPPERTEST *(no game-specific aspects)*
BILLTEST *(no game-specific aspects)*
ATTRACT
update this list

F. SCHEMATIC OF GAME STATE PROGRESSION

Add discusssion and also update *Figure 11-3*

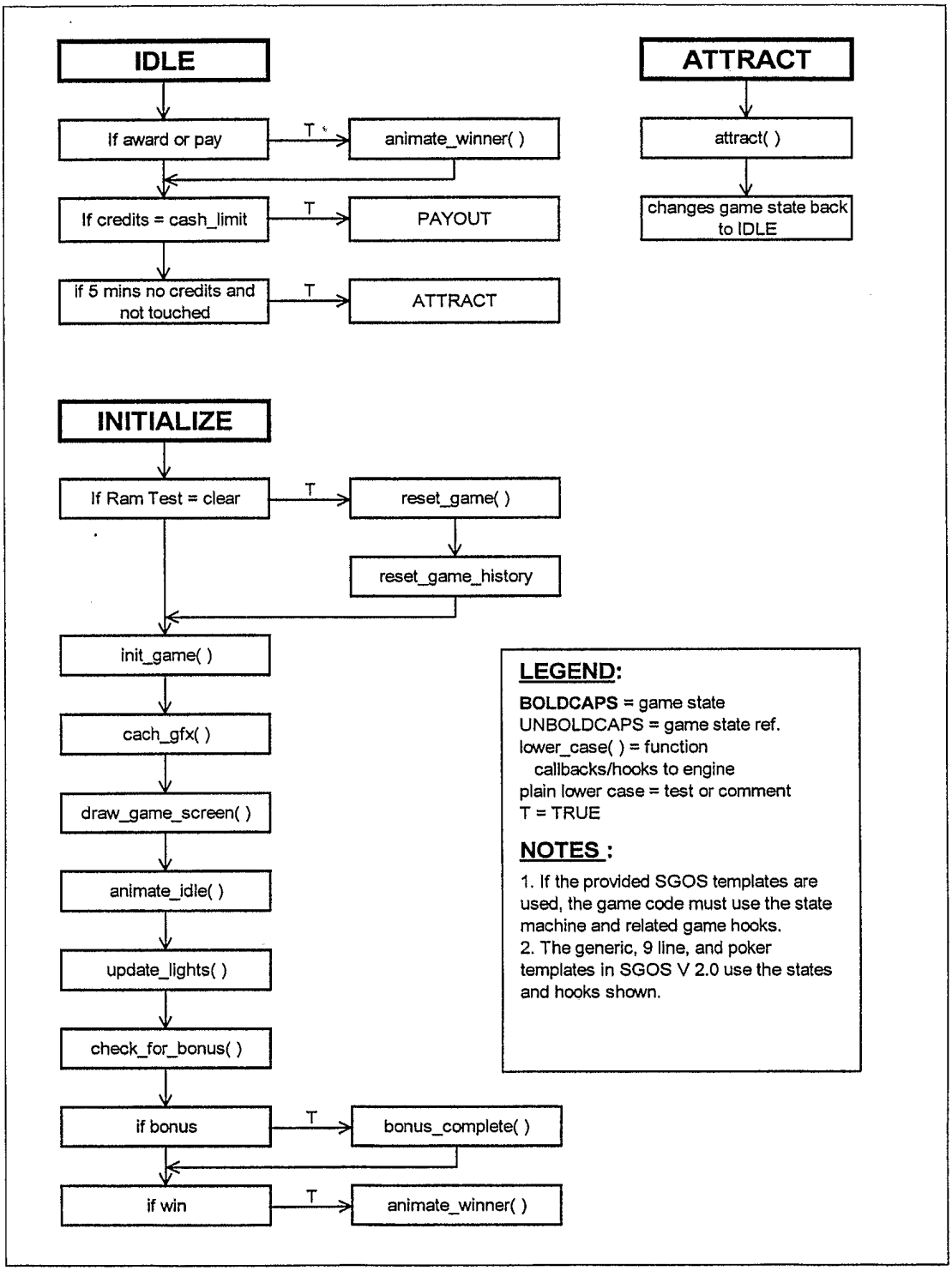


Figure 11-3 – State Machine and Related Game Hooks

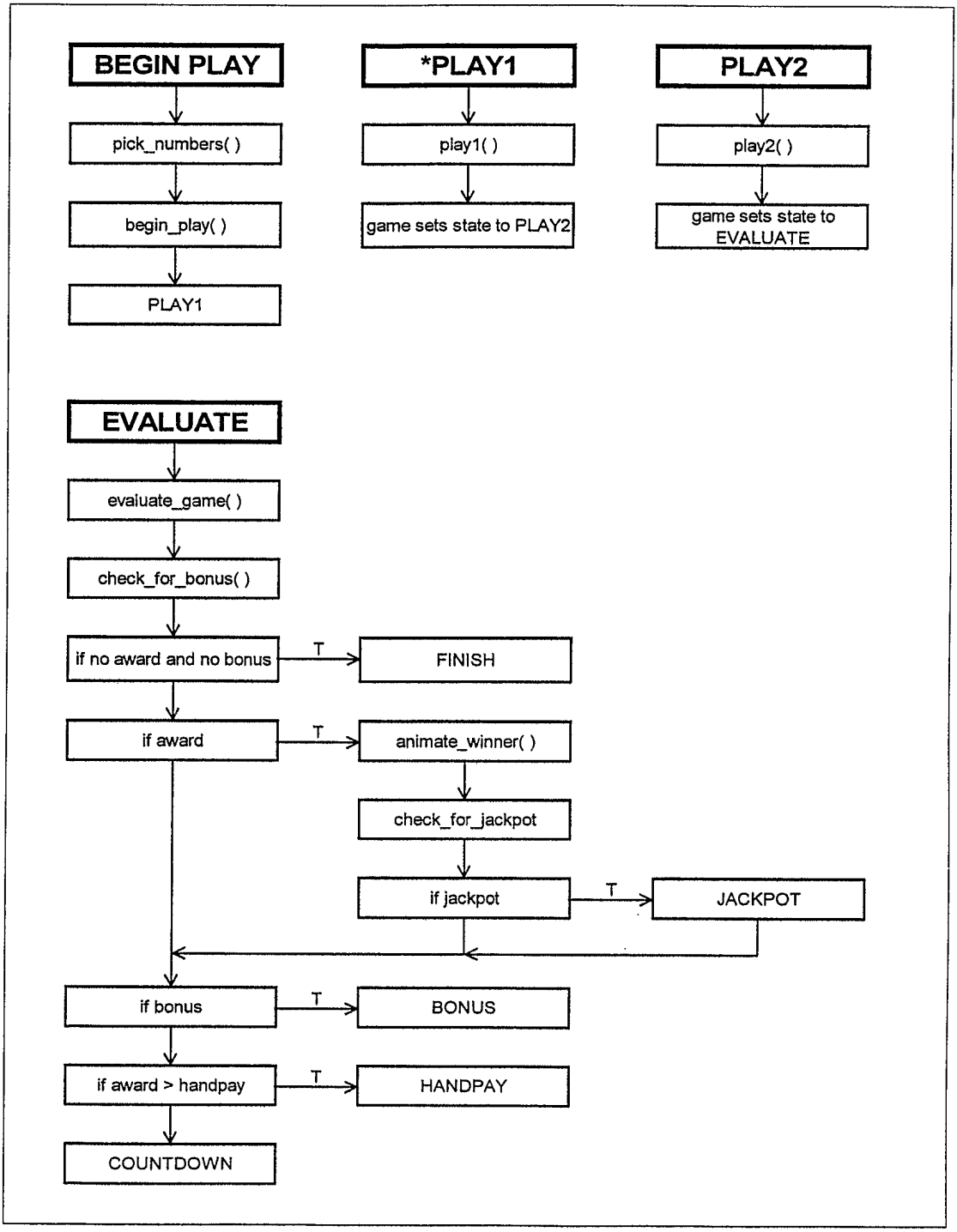


Figure 11-3. State Machine and Related Game Hooks — Continued

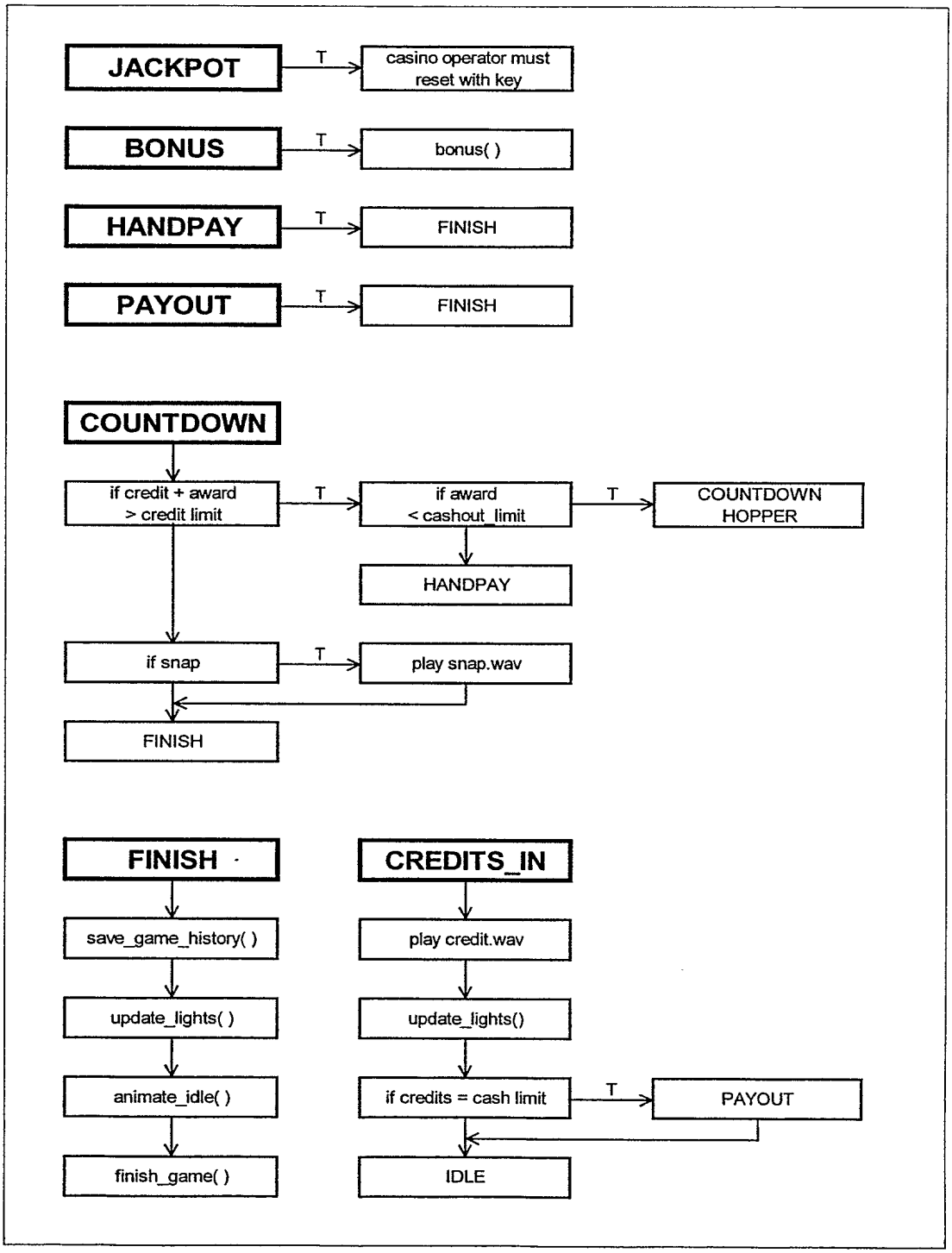


Figure 11-3. State Machine and Related Game Hooks - Continued

CHAPTER 12 – INITIALIZATION (.OTI) FILE

A. BASIC SETTINGS

B. DON'S NEW SECTION

C. ETC.

CHAPTER 13 – MULTIGAME SETUP

A. MULTIGAME CONSIDERATIONS

game_register

game_load

game_inprogress

game_getcurrentstate

game_setcurrentstate

game_setdenomination

B. NAMING OF FILES

C. .OTI INITIALIZATION

CHAPTER 14 — GAME DEVELOPMENT TOOLS

A. C COMPILER

Different compilers may use different instructions. To ensure consistency with SGOS and the Makefiles, stick with the popular gcc C compiler. A copy is included with SGOS.

B. MAKEFILE

The **make** tool and related **Makefile** will be familiar to Linux and UNIX users. **make** is a very powerful way to manage and update your game projects. The **Makefile** tells **make** how to set up dependencies and compile your files. Once you set up the **Makefile** for a project you can quickly recompile and retest your program as you make modifications.

Appendix B explains more about how to set up your **Makefile**, and provides an example which covers the basics you need to build a game in SGOS. However, the appendix only scratches the surface of a tremendously powerful tool, and you may want to use additional features. If you do not already have one, a good Linux programming book is highly recommended.

If you are new to make and the **Makefile**, the tutorial examples will provide good practice in using them to build simple games. *File Listing 16-2* in the first tutorial chapter starts you out with a **makefile** for a single file "game" that simply draws a "Hello World" text string. In the five tutorial chapters that follow you use the makefile several more times to add graphics and other files.

C. CREATION OF REEL STRIPS TOOL

The Makestrips utility will turn a list of par-sheet (tab-delimited text) files into C source code arrays. This Python Script tool is listed and documented in *Appendix H*. The file is included on the SGOS CD-ROM. Once you see how Makestrips works, you might prefer to devise your own tool to set up reel strips.

D. GRAPHICS CONVERSION TOOL

You may have numerous graphics files for your game, especially if you have complex animations. Your game must include each graphic icon as an array in XPM format. The included Python Script tool **convgfx.py** provides a simple way convert your graphics all at once and place them into one graphic icon file. Help in using this tool and further organizing your icons can be found in the following places in this manual:

- *Chapter 7* tells how to use **convgfx.py** and suggests ways to further arrange your icons.
- *Chapter 17* includes a simple example that converts a single icon to an XPM array.
- *Chapter G* provides a file listing of **convgfx.py**.

As with the other SGOS tools, you may want to create a graphics conversion tool of your own that best suits your design approach.

E. DEBUGGING TOOLS

1. gbd Linux Tool

gbd is a popular and powerful Linux tool. It is highly recommended.

2. sys_debug()

sys_debug() is an API call included with SGOS that allows you to put debug statements in your code, and they will be included in **debug.out**. ***is name changed?***

You set debug preferences in the .oti file as follows:

```
debug : {
   output : file;
   options : {all;}
}
```

In this example all will yield copious results for all declared variables. **What are other .oti options?***

You use printf-style formatting to call out each occurrence you want to see in the **debug.out** file with

```
void sys_debug(char* format, ...).
```

You pass a format string followed by variables, if any.

The tutorial in *** gives a helpful example that sets up debug items with extra formatting to help certain timer actions stand out in the generated **debug.out** file.

3. debug.warn and dbug.out

On your development computer (but not on a target machine) SGOS creates **debug.out** and **debug.warn** in the current directory. They are log files created if the game is run with a library compile with the debug option turned on.

debug.warn will let you know of any type mismatches getting or setting variables in nvram, such as passing a character to a variable set as an integer.

debug.out provides a text file of results (e.g. gets and sets from RAM, all events on queue, etc.) for the options set by the user. Programmer can set the level of debug output.

CHAPTER 15 – BUILDING A GAME

A. BUILDING A GAME ON DEVELOPMENT PLATFORM

make, Makefile, etc

B. TESTING CONSIDERATIONS

maybe not

C. BUILDING A GAME ON A TARGET MACHINE

Ref Ch xx hdwr sol'n

Gen notes

IV. API TUTORIAL

CHAPTER 16 — DISPLAYING TEXT

A. OVERVIEW

This chapter creates a simple "Hello World" program using SGOS library calls. The tutorial will include the following SGOS tools and concepts:

- Using SGOS API calls
- Building an operable "game" with the Linux **make** utility
- A few basic guidelines about managing files for an SGOS game

B. ASSEMBLE NEEDED FILES TO RUN SGOS

In your installed **sgos** directory, create a directory named **app.tutorial** with the **mkdir** command:

```
[sgos]$ mkdir app.tutorial
```

TIP... Refer to *Chapter 10-A, File Tree* and *Figure 10-1 – SGOS File Tree* to see how directories and files are arranged in SGOS. *III. GAME DEVELOPMENT* will cover complete file setup requirements for a game.

To successfully run the example you will need to include several files in your new directory. The following files must be in **[your path]/sgos/app.tutorial** for Example 1:

1. **Makefile** used to build the game
2. **example1.c** the Hello World file you just created
3. **game.state** used here only to launch SGOS
4. **impact.ttf** truetype font
5. **local.oti** configuration file for running SGOS on your desktop computer

Locate these files (again refer to the file tree) and copy them to your new **app.tutorial** directory.

WARNING! Note that **local.oti** replaces **start.oti** for your desktop development computer. **local.oti** disables hardware such as the bill handler and coin hopper. If they are not disabled your computer may crash when the program looks for them.

C. USING GFX FUNCTIONS FROM THE USERAPI

This first example is a very simple "Hello World" screen. The code should look familiar if you are an experienced C programmer.

You will need **example1.c** as shown in *File Listing 16-1.* Copy it from the CD-ROM to your **app.temp** directory, or type it in directly using any text editor.

Line 1 includes the header file **userapi.h**, which defines all of basic API functions needed for SGOS programming. *Chapter 5* provides an overview of the available routines. *Appendix C*

File Listing 16-1: example1.c

```
1   #include "userapi.h"
2   void initialize(void){
3       gfx_clearscr(0);
4       gfx_setfont("impact.ttf",72);
5       gfx_drawstring(100,100,"Hello World",RGB(255,255,255),-1);
6   }
```

This file is included in the SGOS installation CD-ROM tutorial directory.

gives a complete listing with explanations.

The remaining code in **example1.c** defines **initialize()**, which is the first function the game engine will call after loading the module. The **gfx** functions are API routines, used as follows:

Line 3 **gfx_clearscr( )** clears the screen buffer to white. Note that **0** is equivalent to **RGB(0,0,0)** in the RGB color convention.

Line 4 **gfx_setfont( )** sets the current truetype font and point size. Each font you specify must be in your program's directory. The font and point size will stay the same until you change it with another **gfx_setfont( )**.

Line 5 **gfx_drawscreen( )** puts the string "Hello World" on the screen buffer.

100, 100 locates the lower left corner of the string at (x, y) = (100, 100) on the 800 x 600 pixel screen.

RGB(255,255,255) defines black as the font color. **RGB(r,g,b)** is a macro function in the SGOS library.

TIP... The last parameter of **gfx_drawscreen()** is an RGB background color. **-1** is shorthand which SGOS converts to a transparency. Likewise, **0** is shorthand for **RGB(0,0,0)**, which is white.

Chapter 7-G, Colors Reserved for Transparency explains how SGOS handles RGB colors and transparency.

D. Using Make and the Makefile

If you are new to Linux, you will soon find that the **make** utility is a powerful tool to build, compile, and manage your programs as you develop them. *Chapter 14-B, Makefile* provides an overview, and you can learn more about **make** in a Linux programming book.

1. Creating the Makefile for example1.c

File Listing 16-2 shows the generic **Makefile** you will need to create projects for all the tutorial examples. You can copy this tutorial version of **Makefile** from the tutorial directory of the CD-ROM **app.temp** diectory or type it from scratch using any text editor.

Note that lines 13 and 14 of **Makefile** refer to **example1.o**, the object file you will create for the Hello World example. For the other tutorial examples, you will change these lines to **example2.o**, etc. and add any other needed files before you build the example game.

Refer to the Makefile listing in Appendix B for a review of syntax and dependency rules.

File Listing 16-2: Makefile for example1.c

```
1   # Makefile for the examples
2   .SUFFIXES: .cpp .c .so
3   DEBUG=-g
4   SGOS=${shell ./locate sgos}
5   INCLUDE=-I.. -I$(SGOS)
6   .c.o:
7       gcc $(INCLUDE) -c -Wall -fPIC -o $@ $<
8   .cpp.o:
9       g++ $(INCLUDE) $(DEBUG) -c -o $@ $<
10  .o.so:
11      gcc -shared -o $@ $<
12  all: game.so
13  game.so: example1.o
14      gcc -shared -o game.so example1.o
```

This file is included in the SGOS installation CD-ROM tutorial file.

Line 13 **game.so: example1.o** says that **game.so** is dependent on **example1.o**.

Line 14 **gcc -shared -o game.so example1.o** is a command to make a shared object out of **example1.o** and name it **game.so**.

When **make** looks at this **Makefile**, it will first try to make **all**. It will next look at the rule for **all** and see that it first needs to make **game.so**. Further rules tell **Make** that it needs **example1.o**, and to make **example1.o** out of **example1.c**. The final result is a new **game.so**.

2. Compiling the Hello World Program

To run the program you must first compile it using **make**. The build process is as follows:

```
[app.tutorial]# make
gcc -I.. -I../framework -c -Wall -fPIC -o test.o test.c
gcc -shared -o game.so test.o
```

As noted above, the program will be compiled as a shared object file named **game.so**. If you are new to Linux, the shared object file is analogous to a Windows **dll** file. When the module loads, the first function it will execute is **initialize()**.

E. Running the Program

To run **example1.so**, you must invoke SGOS by running **start**; **start** then loads the shared object file **game.so** to run the game. The game will call the function **initialize()**, which you defined in **example1.c**.

To run **start** you must be the "super user" (a Linux term meaning you have access to the root directory). If you are not logged in as root, type **su** and the password to become root. Then run the program as follows:

```
[app.tutorial]# /usr/local/sgos/engine.shfl/start
[svgalib: allocated virtual console #8]
Starting: display modules timer
```

You should see the following screen displays:

1. Blank screen.
2. SGOS startup screen.
3. Background comes up transparent, from your **gfx_drawstring** parameter of **-1**.
4. The words "Hello World" will appear in black, as shown in *Figure 16-1*.

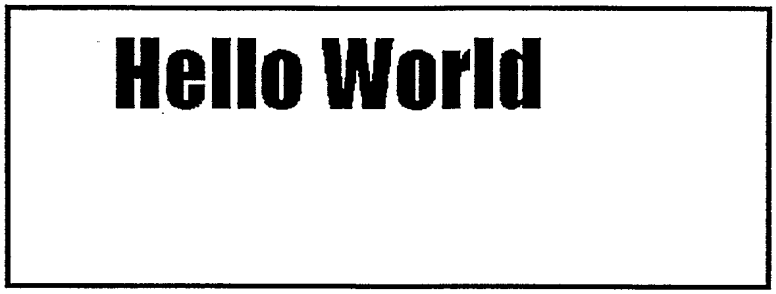


Figure 16-1 – Tutorial: Hello World Screen

Press **q** to quit the program.

F. EXERCISES

You may want to make some changes to some of the parameters in this simple file to observe the response:

1. Change the font (you should have **georgia.ttf** in your **app.temp** directory).
2. Change the font size and color.
3. Change the color of the screen buffer by changing **0** to an **RGB()** value.
4. Replace the **-1** transparency with an **RGB()** value.
5. Move the text around.
6. Center the text on the screen buffer by replacing **gfx_drawstring()** with **gfx_justifystring(0,0,800,600,"Hello World",0,JUSTIFY_CENTER,-1)**. You can also add a new line character with **gfx_justifystring()**, for instance **"Hello\nWorld"**, which **gfx_drawstring()** does not support.

Screen colors are changed for readability.

CHAPTER 17 — DRAWING AN ICON

A. OVERVIEW

This chapter puts a simple graphic icon into the source file and then draws it on the screen. The tutorial will include the following SGOS tools and concepts:

- Graphics conversion tool and XPM format
- SGOS transparency
- Further examples of SGOS **gfx** functions and using the Makefile

B. CONVERTING A GRAPHIC TO XPM FORMAT

You must convert all graphics to the special bitmapped XPM format to display them in SGOS. *Chapter 7* gives details about the conversion tool and how SGOS handles graphics. Copy the file **ball.tif** from the CD-ROM into your **app.tutorial** directory, and use the Python script conversion tool **convgfx.py** as follows:

```
[app.tutorial]# ./convgfx.py ball_gfx
```

Running the script creates the C++ file **ball_gfx.cpp** and the header file **ball_gfx.h**. *File Listing 17-1* shows how the XPM format maps out the graphic of the ball as the array **const char* const ball[]**. In this case it is easy to see the shape of the ball since the graphic has only two colors, represented by the hex characters . and **9**. Although it looks egg-shaped as represented by the characters, it will be round when the actual pixels show up on the screen.

convgfx.py defines each graphic array as the name of the file it came from, except that the file suffix, in this case **.tif**, is dropped. As a result this graphic array is named **ball**. **convgfx.py** is fully listed in *Appendix G*. You can modify it or write your own script if you like, as long you generate the needed **.c** and **.h** files.

C. USING SGOS GFX FUNCTIONS TO DISPLAY A GRAPHIC

You can copy *File Listing 17-2* from the SGOS CD-ROM **app.temp** directory or type it directly.

This file looks like **example1.c**, except that you must add the **ball_gfx.h** header file to include the **ball** graphic, and use different SGOS graphics functions as appropriate to display the icon.

Only lines 2, 5, and 6 have changed, with the following effects:

Ln 1-2 Again include **userapi.h**, plus the file **ball_gfx.h**. created by the graphic conversion tool. The header file simply tells the compiler there is a graphic named **ball**.

Line 3 Again, **initialize()** will be the first function executed upon loading.

Line 4 **gfx_clearscr( )** clears the screen to black. **0** is equivalent to **RGB(0,0,0)**

Line 5 **gfx_drawiconXPM()** puts the graphic onto the screen. SGOS calls graphics "icons," hence the name **drawiconXPM** for the function that draws a graphic of type XPM created by **convgfx.py**. The number pair are the coordinates of the upper left corner

File Listing 17-1: ball_gfx.cpp

```
1    #include "ball_gfx.h"
2    const char* const ball[] = {
3    /* width height num_colors chars_per_pixel */
4       "    50    50      2            1",
5    /* colors */
6       ". c #00aa00",
7       "9 c #ff00ff",
8    /* pixels */
     "999999999999999999999999999999999999999999999999",
     "999999999999999999999999999999999999999999999999",
     "999999999999999999999999999999999999999999999999",
     "99999999999999999.............9999999999999999999",
     "9999999999999.................99999999999999999",
     "999999999999...................9999999999999999",
     "99999999999.....................999999999999999",
     "9999999999.......................99999999999999",
     "999999999.........................9999999999999",
     "99999999...........................999999999999",
     "9999999.............................99999999999",
     "999999...............................9999999999",
     "99999.................................999999999",
     "99999.................................999999999",
     "9999...................................99999999",
     "9999...................................99999999",
     "999.....................................9999999",
     "999.....................................9999999",
     "999.....................................9999999",
    [continued, for the bottom half of the ball...]
 };
```

Code shown at reduced size to show use of **9** *as transparency.*

File Listing 17-2: example2.c

```
1    #include "userapi.h"
2    #include "ball_gfx.h"
3    void initialize(void) {
4        gfx_clearscr(0);
5        gfx_drawiconXPM(200, 200, (char**)ball);
6        gfx_draw3dbar(100, 100, 200, 200, -1, RGB(0, 0, 100), RGB(100, 0, 0), 5);
7    }
```

This file is included in the SGOS installation CD-ROM tutorial file.

of the graphic. The SGOS function **gfx_drawiconXPM()** typecasts **ball** so it will be passed correctly, since it is defined as a constant. The graphic in the file **ball_gfx.cpp** will be joined with this file at link time. The graphic is simply an array of strings as you saw in *File Listing 17-1*.

Line 6 **gfx_draw3dbar(100, 100, 200, 200, -1, RGB(0, 0, 100), RGB(100, 0, 0), 5)** draws a blue and red box.

D. Revise Makefile

You also need to change the Makefile as shown in *File Listing 17-3* to define the dependency of **show_ball.o** on the file **ball_gfx.o**.

File Listing 17-3: Makefile for example2.c

```
1   # Makefile for the examples
2   .SUFFIXES: .cpp .c .so
3   DEBUG=-g
4   SGOS=${shell ./locate sgos}
5   INCLUDE=-I.. -I$(SGOS)
6   .c.o:
7       gcc $(INCLUDE) -c -Wall -fPIC -o $@ $<
8   .cpp.o:
9       g++ $(INCLUDE) $(DEBUG) -c -o $@ $<
10  .o.so:
11      gcc -shared -o $@ $<
12  all: game.so
13  game.so: example2.o ball_gfx.o
14      gcc -shared -o game.so example2.o ball_gfx.o
```

This file is included in the SGOS installation CD-ROM tutorial file.

Lines 1 though 12 are unchanged.

Ln 13-14 **game.so: example2.o ball_gfx.o** tells **make** that the shared object, **game.so**, depends on these two files. Also, you add **ball_gfx.o** to the **gcc** compile instructions.

E. Running the Program

You will see the same SGOS startup screen displays as in the previous example. Then you should see the ball as shown in *Figure 17-1*, below. Press **q** to quit.

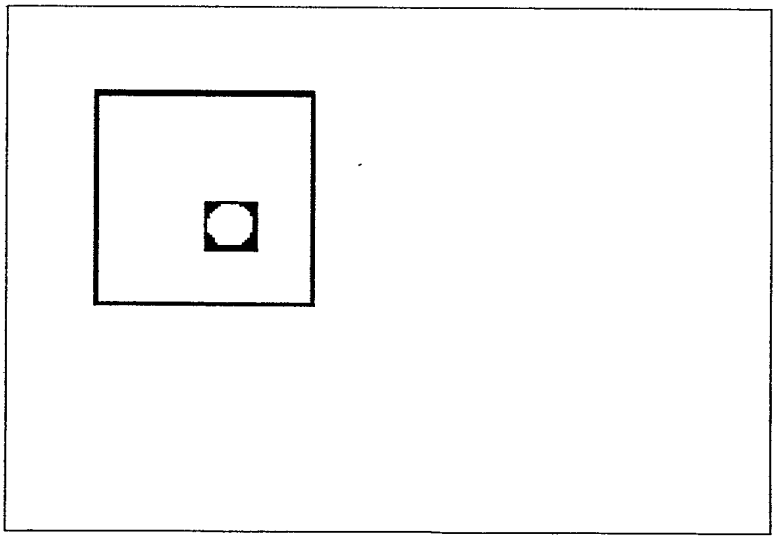


Figure 17-1 – Tutorial: Ball Icon Screen Display

F. Change the gfx Function to Make Transparency Work Correctly

The ball showed up with a bright magenta icon background, because the character **9** in file **ball_gfx.cpp** denotes **RGB(255,0,255)**, pure magenta. The **convgfx.py** conversion tool converted the tiff transparent pixels to this color. You can tell the API you want a transparency by using a graphics function that supports transparency. Replace **gfx_drawiconXPM()** with **gfx_drawtransXPM()** as follows:

```
gfx_drawtransXPM(200,200,(char**)ball,"9")
```

Make this change in your **example2.c** code and then rebuild and run the program. The "transparency" created by **gfx_drawtransXPM()** actually retrieves pixels as needed from SGOS's latest background screen, to effectively create a transparency. Generally, the background screen must be updated for this to work. In this simple example it looked transparent because the screen was still a default black. The next example in *Chapter 18* will show how to update the background screen and introduce other transparency options.

G. Exercises

Try these or other changes to the **example2.c** file:

1. Notice what happens if you change the screen color by passing a different RGB value to **gfx_clearscr()**, e.g. **RGB(255,100,100)**. A black box will appear around the ball. It did not show up before since the screen was black. The frame buffer causes this, and the next chapter will show you more about some important nuances of screen buffering.
2. Try converting other graphics and showing them.
3. Include more than one graphic file, and show multiple graphics at once.

CHAPTER 18 — USING THE FRAME BUFFER AND TIMERS

A. OVERVIEW

This chapter uses the off-screen frame buffer and introduces timers. The tutorial will include the following SGOS tools and concepts:

- Copying the screen buffer to the frame buffer
- Using timers and callback functions for animation
- Role of the event queue
- Example of interaction among nested timers

B. USING THE OFF-SCREEN FRAME BUFFER AND TIMERS

1. Updating the Off-Screen Frame Buffer

The examples so far have drawn directly to the screen buffer. This chapter will show how to use the off-screen frame buffer. The API graphic routines that update and use the frame buffer are at the heart of SGOS animations. This chapter will use **`gfx_copybuffer()`** to update the frame buffer (also called the "background"), and it will use **`gfx_copybufferpart()`** to erase things from the screen buffer.

Recall in the previous tutorial example that **`gfx_drawtransXPM()`** "accidentally" made a correct transparency, only because the default all-black frame buffer happened to match the all-black screen buffer. In this chapter the background will be multicolored, so **`gfx_copybuffer()`** is needed to update the frame buffer.

Chapter 7 gives details about how SGOS uses the frame buffer for transparency and screen updates. *Appendix C* lists and describes the API functions.

2. Using Timers for Animation

SGOS event-driven programming makes extensive use of timers. With every event on its own timer you can easily add and remove items from the screen. Also, by using functions which call themselves back via timers, you can start animations once and not have to worry about them again.

Chapter 18 explains how timers work and their key role in SGOS programming. Basically, a timer is a request to call a function at a certain time. When the timer expires, the request to execute its function is placed on the SGOS event queue. If there are already many functions on the queue or if an executing function is already running, the call to your function will be delayed. So, the actual duration of a timer is only approximate.

A timer is good for only one call. If you want a function to be called repeatedly, you can start as many timers as you need or have the function start a timer to itself before it is done executing.

A timer starts with the API call to **`timer_start()`**. You specify a time delay in milliseconds and a function name (not a pointer). For example, **`timer_start(100,"draw_screen")`**

tells the API to execute the function **draw_screen()** in 0.1 seconds. Note that the name of the function is used, and not a pointer. The function is not called with any parameters, so you would declare **draw_screen** as **void draw_screen(void)**.

Multiple occurrences of a timer can call the same function multiple times. For instance,

```
timer_start(100, "draw_screen");
timer_start(200, "draw_screen");
```

will call **draw_screen()** in 0.1 and 0.2 seconds from now. SGOS has a system limit of 60 timers at any one time. This should not limit your options because calls spend little time in the event queue. The limit is in place because more than 60 timers could create a bottleneck on a typical processor for a target machine.

C. WRITING THE PROGRAM

Copy **example3.c** from the CD-ROM into your **app.tutorial** directory. It should look like *File Listing 18-1.*

This example will give you a good taste of several timers going on at once. Timers and their callback functions are a crucial part of SGOS programming. Once you get used to them, you will find they often simplify your code requirements. Rather than having to tightly control when everything happens, you can launch several different types of actions with no worry they will collide. Each timer will launch its callback function when it comes up the event queue. It does not care what the other callbacks are doing—even if another function fails, it will keep going!

Tutorial **example3.c** uses timers, **for** loops, and a clever incrementing of colors to make a lot happen on the screen. You may want to make various changes to the timers and **gfx** functions to see the effects.

Ln 1-2 Include **userapi.h** and **ball_gfx.h** graphic. Leave the ball in the program, since the next program, which builds on this one, will use it.

Ln 3-8 Define several colors and dimensions.

Ln 9-13 Declare several prototypes: **draw_screen()** draws the checkered screen; **counter()**, **flasher()**, **blinker()** and **blinker2()** are the timer callback functions.

Ln 14-15 **count** and **flash** are global variables used by **counter()** and **flasher()**, respectively.

Ln 16-22 **initialize()** first draws the screen, then continues with these steps:

Copies the screen buffer into the frame buffer with a call to **gfx_copybuffer()**.

Initializes the global variables

Starts two timers, one to call **counter()** in 0.1 seconds and one to call **flasher()** in 0.75 seconds.

Ln 23-34 **draw_screen()** first clears the screen buffer to black, then fills in the entire screen buffer with a grid of boxes, using two **for** loops, below.

ROT_COLOR cycles through some colors by incrementing the color integers.

Ln 35-38 The **counter()** function performs the following:

File Listing 18-1: example3.c

```
1   #include "userapi.h"
2   #include "ball_gfx.h"

3   #define COUNTER_COLOR       RGB(255,0,0)
4   #define FLASHER_COLOR       RGB(0,0,0)

5   #define SCREEN_WIDTH        800
6   #define SCREEN_HEIGHT       600

7   #define BASE_COLOR          0x1dab
8   #define ROT_COLOR(a)        (((a<<1) + (a>>14)) & 0x7fff)

9   void draw_screen(void);
10  void counter(void);
11  void flasher(void);
12  void blinker(void);
13  void blinker2(void);

14  int count;
15  int flash;

16  void initialize(void) {
17      draw_screen();
18      gfx_copybuffer(GFX_SCREENBUFFER,GFX_BACKGROUNDBUFFER);

19      count = 0;
20      flash = 0;
21      timer_start(100,"counter");
22      timer_start(750,"flasher");
    }
23  void draw_screen(void) {
24      int x,y;
25      int color;

26      gfx_clearscr(RGB(0,0,0));
27      color = BASE_COLOR;
28      for(y=0;y<=SCREEN_HEIGHT-50;y+=50){
29          for(x=0;x<=SCREEN_WIDTH-50;x+=50){
30              gfx_drawbar(x,y,50,50,color);
31              color = ROT_COLOR(color);
32          }
33      }
34  }
35  void counter(void) {
36      count++;
37      gfx_setfont("georgia.ttf",36);
38
    gfx_copybufferpart(100,100,200,50,100,100,GFX_BACKGROUNDBUFFER,GFX_SC
    REENBUFFER);
39  gfx_justifystring(100,100,200,50,ideci(count,4),COUNTER_COLOR,
40              JUSTIFY_CENTER,-1);
41      timer_start(100,"counter");
42  }
```

File Listing 18-1: example3.c (continued)

```
43  void flasher(void) {
44      flash = 1 - flash;/* toggle flash */
45      if(flash){
46          gfx_setfont("impact.ttf",72);
47          gfx_justifystring(0,300,800,300,"FLASH!",FLASHER_COLOR,
48                  JUSTIFY_CENTER,-1);
49          timer_start(50,"blinker");
50      }else{
51
    gfx_copybufferpart(0,300,800,300,0,300,GFX_BACKGROUNDBUFFER,GFX_SCREE
    NBUFFER);
52      }
53      timer_start(750,"flasher");
54  }

55  void blinker(void) {
56      if(flash){
57          gfx_drawtransXPM(525,175,(char**)ball,"99");
58          timer_start(50,"blinker2");
59      }
60  }

61  void blinker2(void) {
62
    gfx_copybufferpart(525,175,50,50,525,175,GFX_BACKGROUNDBUFFER,GFX_SCR
    EENBUFFER);
63      timer_start(50,"blinker");
64  }
```

This file is included in the SGOS installation CD-ROM tutorial file.

Increments the global variable count.

Sets the current font to 36-point Georgia.

Enlists **gfx_copybufferpart()** to copy a rectangle from the frame buffer to the screen buffer. The first four arguments define a rectangle in the frame buffer by specifying top left corner, width and height. The next two arguments locate the rectangle on the screen, based on the top left corner of the rectangle, and the last two note the source and destination buffers.

Ln 39-40 **gfx_justifystring()** displays the count on the screen. **ideci()** is a utility function in **userapi.h** that converts an integer to a string. The **4** argument to **ideci()** is the number of digits from the least significant digit to display (e.g. 23456 would be displayed as 3456). It passes back a **char** pointer to a global buffer which is valid until the next call to **ideci()**. **gfx_justifystring()** is first given a rectangle (as top left corner, width and height), the string to display (given by **ideci()**), the text color, the justification of the string in the box (in this case it is centered both vertically and horizontally), and finally the background color, which is **-1** for transparent.

Line 41 Finally, **counter()** starts another timer to call itself back just before it exits.

Ln 43-54 The **flasher()** function is similar to **counter**. It first toggles its global variable which tells whether **flasher()** is flashing on or flashing off.

If it is flashing on, it sets the font to 72-point Impact and draws its string centered in the bottom of the screen. It then starts a timer to **blinker()**.

Otherwise, **flasher()** erases the text from the screen buffer with a call to **gfx_copybufferpart()**.

In either case, it then starts a timer to call itself back.

Ln 55-60 The **blinker()** function first checks to see if **flasher** is on or off by checking the global variable **flash**. If **flasher()** has displayed text, **blinker()** draws the ball icon and then starts a timer to call **blinker2()**.

Ln 61-64 The **blinker2()** function erases the ball icon by copying the corresponding rectangle from the frame buffer, and if **flasher()** is still displaying **flash**, starts a timer to call **blinker()**.

D. Minor Changes to Makefile and Compile

Make and compile the program. Change the filename to **example3.c**. Otherwise this will just be a simple update with no new directives. Although this program does not use the ball graphics, leave the ball in the program since the next example will use it.

E. Run the Program

Run the program the same as in the previous examples. After the startup screen you should see the screen filled with a checkered pattern, and the text animations should flash and count, as shown in *Figure 18-1*.

Unlike the two previous examples, which gave static screen results, **example3.c** will keep blinking and flashing until you quit.

Press **q** to quit.

F. Exercises

Make these adjustments and any others that look interesting. See how your program responds:

1. change **flasher()** so that it is on the screen for .75 seconds and off for .5 seconds.
2. add a function called **mover()** which moves a ball across the screen. Have it cued to begin when **count==300**. Have it end when **count==500**. Continue in this way; turn on when **count%300==0**, turn off when **count%300==200**.

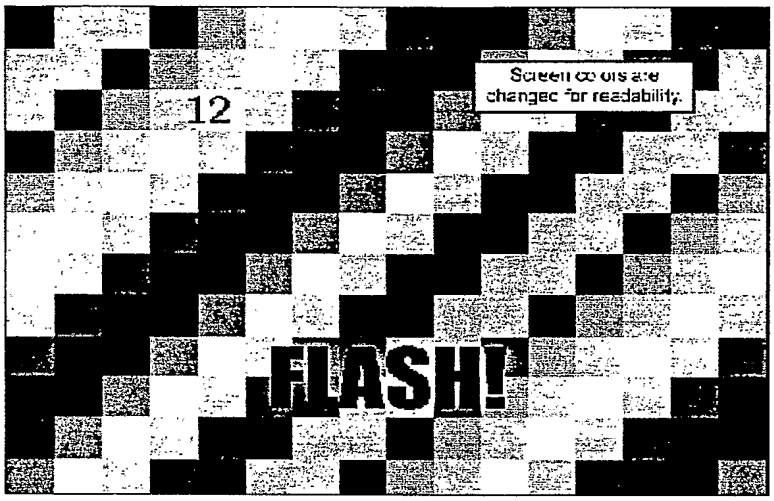


Figure 18-1 – Tutorial: Frame Buffer with Counter

The next tutorial example will add buttons to launch an animation and modify its mission while it runs.

CHAPTER 19 — ADDING BUTTONS

A. OVERVIEW

This chapter adds buttons and other new features to **example4.c** from *Chapter 20*, to give you practice with additional SGOS functions and design approaches. The tutorial will include the following SGOS tools and concepts:

- Add buttons for on/off and reset actions
- Add sound
- Add a ball direction component to show further button and timer nuances
- Use the frame buffer differently to reduce flicker

B. USING SGOS BUTTONS

In this chapter you will add several buttons to launch or modify program actions. The three SGOS button types and "hot spot" routine all work in the same way. The example will use the simplest button type, which contains a simple text string. To learn more about the role of buttons and their API routines, refer to *Chapter 6-G, Button Events* and *Appendix C.*

If the target machine will have mechanical buttons, they can make calls to the same functions. You define hardware lights and switches (panel layout) in the initialization file **start.oti**. See *Appendix D* for details.

WARNING! Do not put buttons in the off-screen frame buffer. As a general rule, make sure that you add buttons to the screen buffer after you use **gfx_copybuffer()**. If you copy a button from the background to the screen you may get a picture of a button with no functionality.

C. WRITING THE PROGRAM

Copy **example5.c** into your **app.tutorial** directory. It should look like *File Listing 19-1.*

This example builds on the previous tutorial in *Chapter 20*. Some of the previous routines are divided into several more functions so each can be distinctly called. The three buttons you add will start, stop, or reset the balls.

First, a brief overview of what the **example5.c** program does:

1. Builds the checkered screen with buttons.
2. When player presses "Go" button:
   a. Starts timer with callback to animate the balls
   b. Plays a button sound
   c. Keeps animation routine going with a timer and callback to itself
   d. Due to the "gravity" function, balls gradually collect at bottom of screen
3. When player presses **STOP!** button:
   a. Kills all timers, which stops all balls
   b. Plays a button sound

File Listing 19-1: example5.c

```
1   #include "userapi.h"
2   #include "ball_gfx.h"

3   #define BALL_WIDTH  50
4   #define BALL_HEIGHT  50

5   #define SCREEN_WIDTH  800
6   #define SCREEN_HEIGHT  500

7   #define GRAVITY  3
8   #define NUM_BALLS  1

9   #define BASE_COLOR  0x1dab
10  #define ROT_COLOR  (a)(((a<<1) + (a>>14)) & 0x7fff)

11  typedef struct {
12      int x,y;
13      int x_vel,y_vel;
14  } ball_struct;

15  void initialize(void)
16  void draw_screen(void);
17  void draw_grid(void);
18  void init_balls(void);
19  void anim_balls(void);
20  void calc_pos(ball_struct * b);
21  void draw_balls(void);
22  void erase_balls(void);
23  void draw_telemetry(void);
24  void go_button(void);
25  void stop_button(char * name);
26  void reset_button(void);
27  void refresh_screen(void);

28  ball_struct balls[NUM_BALLS];
29  int step;

30  void initialize(void) {
31      draw_screen();

32      step = 0;
33      init_balls();
34      timer_start(100,"draw_telemetry")}

35  void draw_screen(void) {
36      gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
37      gfx_clearscr(RGB(0,0,0));
38      draw_grid();
39      gfx_setgraphicscontext(GFX_SCREENBUFFER);
40      gfx_copybuffer(GFX_BACKGROUNDBUFFER,GFX_SCREENBUFFER);
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 19-1: example5.c (continued)

```
41      /* make the buttons */
42      makebutton1("GO!",10,510,100,50,RGB(0,255,0),"go_button");
43      makebutton1("STOP!",10,570,100,30,RGB(255,0,0),"stop_button");
44      makebutton1("Reset",120,510,50,90,RGB(255,255,0),"reset_button");
45      setbuttonfont("GO!", "impact.ttf", 30);
46  }

47  void draw_grid(void){
48      int x,y;
49      int color;

50      color = BASE_COLOR;
51      for(y=0;y<=SCREEN_HEIGHT-50;y+=50){
52          for(x=0;x<=SCREEN_WIDTH-50;x+=50){
53              gfx_drawbar(x,y,50,50,color);
54              color = ROT_COLOR(color);
55          }
56      }
57  }

58  /** button callbacks **/

59  void go_button(void){
60      timer_start(100,"anim_balls");
61      sound_play("button.wav");
62  }

63  void stop_button(char * name){
64      sys_debug(">>>> stop button: %s",name);
65      timer_kill("anim_balls");
66      sound_play("button.wav");
67  }

68  void reset_button(void){
69      init_balls();
70      refresh_screen();
71      sound_play("button.wav");
72  }

73  /** telemetry display **/

74  void draw_telemetry(void){
75      char buf[60];

76      text_printf(buf,"step %d",step);
77      gfx_setfont("georgia.ttf",12);
78      gfx_copybufferpart(500,510,300,30,
    500,510,GFX_BACKGROUNDBUFFER,GFX_SCREENBUFFER);
79      gfx_drawstring(510,530,buf,RGB(255,255,255),-1);

80      timer_start(900,"draw_telemetry");
81  }
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 19-1: example5.c (continued)

```
82  /** ball animation **/

83  void init_balls(void) {
84      int i;
85      for(i=0;i<NUM_BALLS;i++){
86          balls[i].x = rnd_get_number(SCREEN_WIDTH - BALL_WIDTH);
87          balls[i].y = rnd_get_number(SCREEN_HEIGHT - BALL_HEIGHT);
88          balls[i].x_vel = rnd_get_number(200) - 100;
89          balls[i].y_vel = rnd_get_number(200) - 100;
90      }
91  }

92  void anim_balls(void) {
93      int i;
94      gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
95      erase_balls();
96      for(i=0;i<NUM_BALLS;i++){
97          calc_pos(&balls[i]);
98      }
99      draw_balls();
100     step++;
101     gfx_setgraphicscontext(GFX_SCREENBUFFER);  //unneeded
102
    gfx_copybufferpart(0,0,SCREEN_WIDTH,SCREEN_HEIGHT,0,0,GFX_BACKGROUNDB
    UFFER,GFX_SCREENBUFFER);
103     timer_start(100,"anim_balls");
104 }

105 void draw_balls(void) {
106     int i;
107     for(i=0;i<NUM_BALLS;i++){
108         gfx_drawsprite(balls[i].x, balls[i].y,(char**)ball);
109     }
110 }

111 void erase_balls(void) {
112     draw_grid();
113 }

114 void refresh_screen(void) {
115     gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
116     draw_grid();
117     draw_balls();
118     gfx_setgraphicscontext(GFX_SCREENBUFFER);
119
    gfx_copybufferpart(0,0,SCREEN_WIDTH,SCREEN_HEIGHT,0,0,GFX_BACKGROUNDB
    UFFER,GFX_SCREENBUFFER);
120 }

121 void calc_pos(ball_struct * b) {
122     b->x += b->x_vel;
123     b->y += b->y_vel;
124     b->y_vel += GRAVITY;
```

File Listing 19-1: example5.c (continued)

```
125 if(b->x <= 0){
126         b->x = 0;
127         b->x_vel = -(b->x_vel*8/10);
128     }else if(b->x >= SCREEN_WIDTH - BALL_WIDTH){
129             b->x = SCREEN_WIDTH - BALL_WIDTH;
130             b->x_vel = -(b->x_vel*8/10);
131     }
132     if(b->y <= 0){
133         b->y = 0;
134         b->y_vel = -(b->y_vel*8/10);
135     }else if(b->y >= SCREEN_HEIGHT - BALL_HEIGHT){
136             b->y = SCREEN_HEIGHT - BALL_HEIGHT;
137             b->y_vel = -(b->y_vel*8/10);
138     }
139 }
```

This file is included in the SGOS installation CD-ROM tutorial file.

4. When player presses **Reset** button:
   a. Re-creates all balls from scratch
   b. Plays a button sound

The following review of the code in **example5.c** focuses on changes from the previous example in *Chapter 20*:

Ln 3-10 The definitions are unchanged from the last example, except that the screen height is reduced from 600 to 500, to allow room for the black bar at the bottom.

Ln 29-34 The **initialize()** routine still creates the initial ball positions and velocities. The previous example drew directly to the screen and used **gfx_copybuffer()** to set the frame buffer. Now you will use the frame buffer as an intermediate step, as explained below in **draw_screen()**.

initialize() has a new step to start a timer for the telemetry string on the bottom of the display.

Ln 35-46 The **draw_screen()** routine uses the frame buffer this time as a true buffer, as follows:

gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER) lets you draw directly into the frame buffer. Nothing will display on the screen.

gfx_clearscreen() clears the frame buffer.

draw_grid() is now a separate function (see below). In the last example it was part of **initialize()**.

After drawing the grid, **gfx_setgraphicscontext(GFX_SCREENBUFFER)** sets the graphics context back to the screen.

gfx_copybuffer() copies the frame buffer onto the screen buffer.

makebutton1("GO!", 10, 510, 100, 50, RGB(0, 255, 0), "go_button") creates a button named **GO!**. This button is located at **(10, 510)**, and has a width of **100** and a height of **50**. **RGB(0, 255, 0)** defines its color as green. The callback function **go_button** is called when the button is pressed.

The parameters for the **Stop** and **Reset** buttons are similar.

setbuttonfont() changes the font and point size of the **GO!** button only. The other two buttons will use the button font default, which is Georgia 8 point.

Ln 47-57 **draw_grid()** is a separate new function to draw the grid of colored boxes. The code is unchanged from the last example when it was a part of the **initialize()** function.

Ln 59-62 **The go_button()** function is called when the **GO!** button is pressed. It simply starts a timer for the ball animation and plays a button sound.

TIP... Your development system probably does not have sound capabilities. In the **local.oti** initialization file, sound must be an included option for **[startup]**. Refer to the file listing in *Appendix D*.

Ln 63-67 The **stop_button()** function does the following:

Prints a debug statement giving the parameter passed. If a callback function passes a parameter, the parameter is the name of the button pressed. You could have set up several buttons that call the **stop_button()** function, but this example has only the one button named **Stop**. So the parameter is always **Stop**.

Deletes all the timers having **anim_balls()** as a callback function. This action freezes the animation.

Plays a button sound.

Ln 68-72 The **reset_button()** function re-runs **init_balls()** and **refresh_screen()**. Then it plays the button sound.

Ln 74-81 The **draw_telemetry()** function does the following:

Allocates a character buffer and writes into it using **text_printf()**, which works just like the standard library function **printf()**.

Sets the current font to **12** point **georgia.ttf**, erases whatever is now in the display spot and draws the string on the screen.

Starts a timer to make a callback to itself. Notice that this is the second timer loop that is drawing to the screen. (The other one is for animation of the balls.) This does not create any problems since they draw to different areas of the screen and don't interfere with each other.

TIP... No standard C calls are permitted in game code, to preserve the integrity of the pre-approved library and game engine. This includes file I/O and the **printf** family. For instance, this example uses **text_printf()** from the SGOS library instead of **sprintf()**.

Ln 83-91 **init_balls()** is now a separate function to set up each ball's location and velocity. The code is unchanged from the last example when it was a part of the **initialize()** function.

Ln 92-104 The function **anim_balls()** is refined from the way it worked in the previous example. The animation logic is still the same, but now you draw to the frame buffer first and then draw to the screen. The steps in **anim_balls** are as follows:

Set the graphics context to **GFX_BACKGROUNDBUFFER**.

Erase all present balls from the frame buffer by redrawing the entire colored grid.

Update the positions of all the balls with **draw_balls()**.

Draw all the balls into the frame buffer.

Set the graphics context back to **GFX_SCREENBUFFER** to draw to the screen.

Copy the entire checkered area with the balls from the frame buffer to the screen. This single screen update greatly reduces flicker.

As its last step, **anim_balls** starts a timer with a callback to itself.

Ln 105-10 **draw_balls()** has become a separate function in the current example. It switches to **gfx_drawsprite()** to handle transparency.

Recall from the last example that the **gfx_drawtransXPM()** transparency mechanism erased overlapping parts of ball images as new ones were added. Since **gfx_drawsprite()** replaces transparent pixels with the pixels from the current buffer (in this case the frame buffer), each new ball placement will put only the ball itself into the current frame buffer. All the transparency of the ball's rectangular border will be replaced with appropriate pixels from the current frame buffer.

Ln 111-13 **erase_balls()** simply redraws the entire colored grid. This complete redraw is a somewhat primitive approach, but it works well for this example. *Chapter 7-M, Double-Buffered Animation,* gives an example of a more advanced graphics updating approach.

Ln 114-20 The **refresh_screen()** function has the following steps:

Sets the graphics context to the **GFX_BACKGROUNDBUFFER**. This approach will copy all changes to the screen at once, giving a nicer presentation.

Erases all the balls by having **draw_grid()** redraw the screen.

Creates new ball positions and velocities with **init_balls()**.

Draws the balls on the screen.

Sets the graphics context back to the **GFX_SCREENBUFFER** so all the new balls can be drawn to the screen at once. Then it copies the rectangle, defined from the upper left corner of the screen and **SCREEN_WIDTH** x **SCREEN_HEIGHT** in size, to the screen.

Ln 121-39 **calc_pos()** is unchanged from before.

D. Make, Compile, and Run the Program

Make and compile the program the same as in the last example. Change the example file name

to **example5.c**. Also, include the sound file **button.wav**.

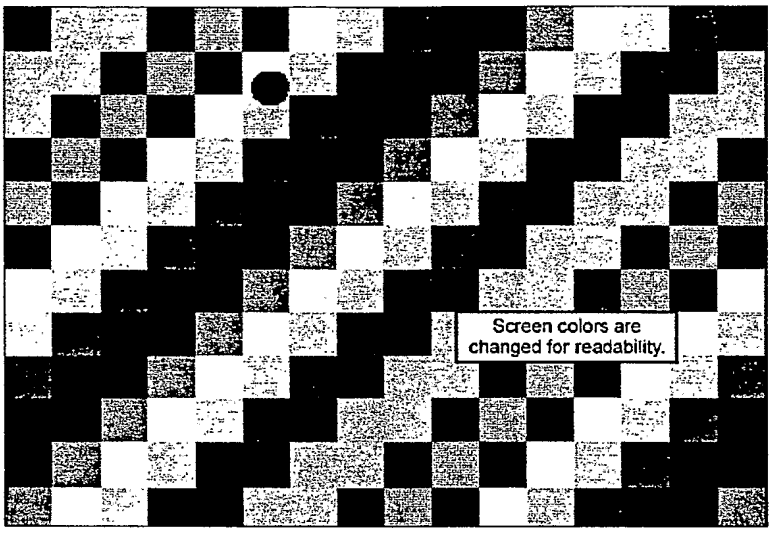


Figure 19-1 – Tutorial: Screen with Buttons

When you run the program you will see a checkered display with a black bar along the bottom, as shown in *Figure 19-1*. There are three buttons: **GO!**, **Stop**, and **Reset**. Click on the **GO!** button to start the animation. Press **Reset** to give the balls a random new position and velocity

If you press **GO!** more than once, it will keep spawning animation timers, and the balls will move faster up to a point. The animation loop moves all balls each time it is called. It does not care which timer loop called it, so more timers will just speed things up.

If you then press **Stop**, all the animation timers will be deleted and the balls will stop.

E. Circumstance Where Callbacks Are Not Stopped by timer_kill()

1. Why the Animation May Not Stop

If a lot of timers are running when you press **Stop**, the animation may fail to fully stop. When many timers are running, it is likely that one or more of them will still be waiting in the event queue. Once a timer times out, only the callback function remains; it is no longer a "timer" and is no longer affected by the **timer_kill()** function.

The more animation timers there are, the greater the chances the **timer_kill()** functions will not stop a timer callback. If there are as many as 30 animation timers, this phenomenon of callbacks slipping through the queue may happen to more than one timer. SGOS does limit the number of timers that can be running at one time (maximum is 60).

The **debug.out** file lists timers by the time remaining on them, in the order of increasing time left. Timers are not associated to numbers, so you cannot track a particular timer through the **debug.out** file. **debug.out** is more useful to see how many timers are executing at one time.

2. A Sure Way to Stop the Animation

There is another way you can set up the on-off switches. Use a global variable as a boolean telling whether the animation is running. Check this global variable in `go_button()`. If the animation is not running, set the flag and start the animation. In this chapter's example, `stop_button()` just sets the flag to false, whereas `anim_ball()` checks each time to see if it should be animating by looking at the flag. If not, the timer is not renewed.

F. EXERCISES

The following exercises show some further nuances of how buttons and timers work in SGOS:

1. Set the constant `NUM_BALLS` to 30. Notice how much smoother the animation is with more balls than the previous example.
2. The example program as written allows many animation timers by repeatedly pressing `GO!` Try pressing `GO!` about 30 times. Press `Stop`, then quit the program and look in `debug.out` for >>>>, the characters at the front of the `stop button` string. Just above the >>>> you will see a listing of most of the 30 timers. After the >>>> there should be a short listing of timers that did not stop before you quit the program. One of these timers is the telemetry timer which you did not kill. All the others are for the animation loop. How many slipped through the `kill_timer()` call on the event queue?

CHAPTER 20 — USING TIMERS TO MOVE AN ICON

A. OVERVIEW

The icon in the previous example flashed and blinked but stayed in one position. This chapter makes the ball icon move. The tutorial will include the following SGOS tools and concepts:

- Using a structure to give the ball position and velocity
- Getting a random number from the SGOS library
- Using the SGOS debug statement
- Using the SGOS "sprite" graphics function to avoid clipping in animations

B. GETTING A RANDOM NUMBER FROM THE SGOS LIBRARY

This chapter will use random numbers to move balls around the screen at varying velocity. SGOS includes its own random number generator. Use the API function

```
rnd_get_number(range)
```

to return a random number between zero and **range**. Refer to *Appendix C* for more about this function.

C. DEBUG SETTINGS

The API call **sys_debug(char* format, ...)** prints the named **format**s to the file **debug.out**. This chapter will use it to find the ball's current position. Refer to *Chapter 14-E, Debugging Tools* for more about SGOS debugging options.

D. WRITING A PROGRAM THAT MOVES AN ICON

Copy **example4.c**, as listed in *File Listing 20-1*, to your **app.tutorial** directory. This example will move a ball around the screen and demonstrate a few important points about transparency. It will again use the ball icon from *Chapter 17*, and will add code to provide and display ball position and velocity. Following is a review of the code in **example3.c**:

Ln 1-2 Include **userapi.h** and **ball_gfx.h** as in the previous examples.

Ln 3-10 Define several constants:
BALL_WIDTH and **BALL_HEIGHT** are the width and height of the ball graphic in pixels.
SCREEN_WIDTH, SCREEN_HEIGHT are set, also in pixels.
GRAVITY is the acceleration of the balls to the bottom of the screen, in no particular units.
NUM_BALLS is the number of balls handled in the program.
Lines 9 and 10 define the **BASE_COLOR** and **ROT_COLOR**, which steps through several colors by incrementing the color character value.

Ln 11-14 Declare a ball structure. The structure contains the ball's position on the screen and its velocity.

Ln 15-20 Create global variables to keep information between timer callbacks:

File Listing 20-1: example4.c

```
1   #include "userapi.h"
2   #include "ball_gfx.h"

3   #define BALL_WIDTH  50
4   #define BALL_HEIGHT  50

5   #define SCREEN_WIDTH  800
6   #define SCREEN_HEIGHT  600

7   #define GRAVITY  3
8   #define NUM_BALLS  1

9   #define BASE_COLOR  0x1dab
10  #define ROT_COLOR  (a)(((a<<1) + (a>>14)) & 0x7fff)

11  typedef struct {
12      int x,y;
13      int x_vel,y_vel;
14  } ball_struct;

15  void initialize(void);
16  void draw_screen(void);
17  void anim_balls(void);
18  void calc_pos(ball_struct * b);

19  ball_struct balls[NUM_BALLS];
20  int step;

21  void initialize(void) {
22      int i;

23      draw_screen();
24      gfx_copybuffer(GFX_SCREENBUFFER,GFX_BACKGROUNDBUFFER);
25      step = 0;
26      for(i=0;i<NUM_BALLS;i++){
27          balls[i].x = rnd_get_number(SCREEN_WIDTH - BALL_WIDTH);
28          balls[i].y = rnd_get_number(SCREEN_HEIGHT - BALL_HEIGHT);
29          balls[i].x_vel = rnd_get_number(200) - 100;
30          balls[i].y_vel = rnd_get_number(200) - 100;
31      }
32      timer_start(100,"anim_balls");
33  }

34  void draw_screen(void) {
35      int x,y;
36      int color;
37      gfx_clearscr(RGB(0,0,0));
38      color = BASE_COLOR;
39      for(y=0;y<=SCREEN_HEIGHT-50;y+=50){
40          for(x=0;x<=SCREEN_WIDTH-50;x+=50){
41              gfx_drawbar(x,y,50,50,color);
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 20-1: example4.c (continued)

```
42              color = ROT_COLOR(color);
43          }
44      }
45  }

46  void anim_balls(void) {
47      int i;
48      for(i=0;i<NUM_BALLS;i++){
49          gfx_copybufferpart(balls[i].x,balls[i].y,
50                  BALL_WIDTH,BALL_HEIGHT,
51
    balls[i].x,balls[i].y,GFX_BACKGROUNDBUFFER,GFX_SCREENBUFFER);
52          calc_pos(&balls[i]);
53          sys_debug(">>> step=%d\tball=%d\tv^2=%d",step,i,
54                  balls[i].x_vel*balls[i].x_vel
55                  +balls[i].y_vel*balls[i].y_vel);
56      }
57      for(i=0;i<NUM_BALLS;i++)
58          gfx_drawtransXPM(balls[i].x,balls[i].y,
59                  (char**)ball,"99");
60      step++;
61      timer_start(100,"anim_balls");
62  }

63  void calc_pos(ball_struct * b) {
64      b->x += b->x_vel;
65      b->y += b->y_vel;
66      b->y_vel += GRAVITY;

67      if(b->x <= 0){
68          b->x = 0;
69          b->x_vel = -(b->x_vel*8/10);
70      }else if(b->x >= SCREEN_WIDTH - BALL_WIDTH){
71              b->x = SCREEN_WIDTH - BALL_WIDTH;
72              b->x_vel = -(b->x_vel*8/10);
73      }
74      if(b->y <= 0){
75          b->y = 0;
76          b->y_vel = -(b->y_vel*8/10);
77      }else if(b->y >= SCREEN_HEIGHT - BALL_HEIGHT){
78              b->y = SCREEN_HEIGHT - BALL_HEIGHT;
79              b->y_vel = -(b->y_vel*8/10);
80      }
```

This file is included in the SGOS installation CD-ROM tutorial file.

balls is the array holding all of the ball structures.
step is a counter of times through the animation loop.

Ln 21-33 The function **initialize()** will be called by SGOS on startup.

Call **draw_screen()** to draw the checkered pattern, then use **gfx_copybuffer()** to copy the entire screen buffer into the frame buffer for use to refresh the display when the ball(s) moves.

Initialize the global variable **step** and each of the **balls[i]** in the balls array. This routine uses random numbers to initialize the ball array. The code uses **rnd_get_number(200) - 100** to generate velocities in the range -100<=x<100.

Now that everything is set up, start a timer to begin the animation.

Ln 34-45 **draw_screen()** does the following:

Clears the screen buffer to black using **gfx_clearscr(RGB(0,0,0)**.
Draws the boxes, using a **for** loop to rotate through colors using the variable **c**.

Ln 46-62 **anim_balls()** does the following:

For each ball in the array: Uses **gfx_copybufferpart** to erase it from the screen by copying the relevant part from the specified source, the frame buffer (the background), to the destination buffer, the screen buffer; then calculates the ball's new position and print a debug statement.

After all balls are erased and updated, **gfx_drawtransXPM()** enlists the **for** loop to draw each ball to the screen.

To keep things going, start a timer to call **anim_balls()** back in .1 second.

Ln 63-80 **calc_pos()** contains nothing related to SGOS. It calculates a ball's new position and velocity, including factors for inelastic collisions with the sides of the screen and gravity.

E. Running the Program

Run the program the same as in the previous examples. After the startup screen you should again see the screen filled with a checkered pattern. A single ball icon will move around the screen, because you inially set **NUM_BALLS** to 1. *Figure 20-1* shows how the screen should look. **example3.c** will slowly run down because the balls have "gravity." Press **q** to quit..

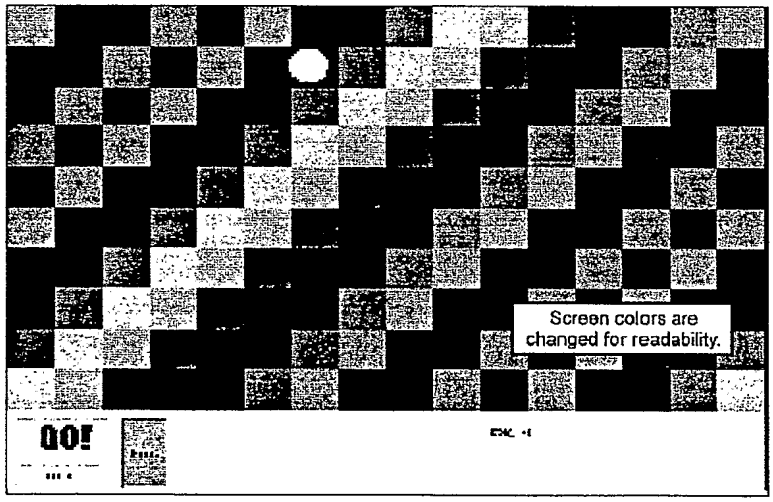


Figure 20-1 – Tutorial: Timers and a Moving Icon

F. Using Sprite to Fix Animation Clipping

If you have not tried it already, increase **NUM_BALLS** and run the program again.

Notice that the ball icons' rectangles clip each other and hide portions of the balls that should be displayed. This happens because **gfx_drawtransXPM()** replaces the transparent pixels in a graphic with the appropriate pixels from the frame buffer. In effect, the **gfx_drawtransXPM()** transparency mechanism erases overlapping parts of any ball images already drawn to the screen.

Using a **sprite** animation function will resolve the clipping because it handles transparency differently. The function **gfx_drawsprite()** will replace the transparent pixels with the appropriate pixels from the current buffer, which in this case is the screen.

The next tutorial example will use **sprite** functions.

G. EXERCISES

Try the following:

1. Experiment with **gfx_drawsprite()** if you want to see its effect. Refer to *Appendix C*.
2. When you increase the number of balls you will see them flicker as they collect near the bottom of the screen. The next examples will show refined ways to update screens to minimize flicker.

CHAPTER 21 — TUTORIAL: USING NVRAM

A. OVERVIEW

This chapter shows how to use the non-volatile RAM (nvram) to preserve data among multiple program modules. The tutorial will include the following SGOS tools and concepts:

- How SGOS lets you interact with nvram
- Using the nvram file in place of nvram hardware on your development computer
- An improved graphics technique
- More buttons to show more interactions among multiple game functions

B. ADDING NVRAM TO PRESERVE DATA

To satisfy security and accounting requirements, all computer-based games of chance must be able to preserve data across executions. This chapter introduces SGOS nvram storage with an example that remembers two structures to store ball positions. *Chapter 9* explains more about how SGOS uses the **game.state** file to handle nvram data. *Appendix C* describes the API nvram routines.

Since your desktop computer will not have nvram hardware, you must use a file called **nvram** in its place. The file **local.oti** disables nvram hardware for development (it also disables the touch screen and other game hardware). Refer to *Appendix D* and *Chapter 2-2, Installing and Configuring SGOS.* While you are developing a game, SGOS will use your **nvram** file as a proxy for the nvram hardware.

C. USE OF MULTIPLE GAME MODULES

Spreading code among different modules can help keep it cleaner. Whenever you load a new module, SGOS first unloads everything from memory in the current module and calls **mod_exit()**, then it loads the new module into memory and calls **initialize()**. **nvram** is the only way you can pass parameters between modules.

Yor must name your game's main module **game.so** for it to work correctly. Other modules can have any name. This chapter creates a separate module for the ball's "gravity" function. The file called **gravity.c** becomes **gravity.so** when compiled.

D. GRAPHICS HANDLING ALTERNATIVES

The example in the preceding chapter used a primitive animation scheme that redrew the entire background image with all the balls each time through the loop. It then copied the entire background image to the screen each time.

You really only need to redraw enough of the background to erase each ball and copy all changes from the background to the screen. However, this example with the multiple balls will stay with the primitive approach. The rectangle tracking scheme for multiple updates would get unwieldy.

Chapter 7-M, Double-Buffered Animation explains how to use a more sophisticated approach for localized screen updates.

Though still elementary, the **draw_telemetry()** function does use a more localized approach for its screen update, which would reduce flicker. It first updates the frame buffer with the original background — solid black — and then draws its new string in the frame buffer. Finally, it copies the rectangle with the changed part of the frame buffer to the screen. In this case the string never moves, the background is very simple, and there are no other overlapping images.

E. CREATE A GAME.STATE FILE

SGOS uses the **game.state** file to store variables and structures during and between games. You will store two structures in **game.state** in this example. *File Listing 21-1* shows the code

File Listing 21-1: game.state file for example6.c

```
1   struct ball_info{
2       char initialized;
3       int num_balls;
4       int step;
5       int moving;
6       int gravity_x CALLBACK(update_gravity);
7       int gravity_y CALLBACK(update_gravity);
8       } Info;
9   struct ball{
10      int x;
11      int y;
12      int x_vel;
13      int y_vel;
14      int handler;
15      } Ball[50];
```

This file is included in the SGOS installation CD-ROM tutorial file.

to store **struct ball_info** and **struct ball**. These two structures store all the ball data that will be needed between modules, so the game can continue as the modules are loaded and unloaded.

F. CREATE MODULE FOR EXAMPLE6.C

As a game gets more complex it is helpful to break it up into several working modules. For the current example you will create two program files that will act as two distinct modules:

- **example6.c** will start, stop, and run most of the ball actions
- **gravity.c** will provide buttons to adjust a directional "gravity" property

The following is a review of the code in **example6.c** (refer to *File Listing 21-2*).

Ln 1-3 These included files should look familiar from the other examples. For **example6.c** you are also adding a second ball, **ball_red_gfx.h**. ***blue?***

Ln 4-11 Most of the defined ball properties also look familiar, but there are two changes. Now you define **MAX_BALLS** instead of **NUM_BALLS** because the new program has but-

File Listing 21-2: example6.c (sheet 1 of 7)

```
1   #include "userapi.h"
2   #include "ball_gfx.h"
3   #include "ball_red_gfx.h"

4   #define BALL_WIDTH  50
5   #define BALL_HEIGHT  50

6   #define SCREEN_WIDTH  800
7   #define SCREEN_HEIGHT  500

8   #define MAX_BALLS  50
9   #define NUM_HANDLERS  2

10  #define BASE_COLOR0x1dab
11  #define ROT_COLOR(a)(((a<<1) + (a>>14)) & 0x7fff)

12  /** the ball structure **/

13  typedef struct {
14      int x,y;
15      int x_vel, y_vel;
16      int handler;
17  } ball_struct;

18  /** prototypes **/

19  void initialize(void);
20  void draw_screen(void);
21  void draw_grid(void);
22  void go_button(void);
23  void stop_button(char * name);
24  void reset_button(void);
25  void remember_button(void);
26  void recall_button(void);
27  void more_button(void);
28  void less_button(void);
29  void gravity_button(void);
30  void draw_telemetry(void);
31  void init_ball(int number);
32  void init_balls(void);
33  void anim_balls(void);
34  void draw_balls(void);
35  void refresh_screen(void);
36  void add_rectangle(int x, int y, int w, int h);
37  void copy_rects(void);
38  void redraw_framebuffer(void);
39  void calc_pos(ball_struct * b);
40  void calc_pos_rnd(ball_struct * b);
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-2: example6.c (sheet 2 of 7)

```
41  /** globals **/

42  ball_struct balls[MAX_BALLS];
43  int step;
44  int num_balls;
45  int is_moving;
46  int gravity_x;
47  int gravity_y;

48  /** main function **/

49  void initialize(void) {
50      draw_screen();

51      step = 0;
52      num_balls = 1;
53      is_moving = 0;

54      nv_getint("Info.gravity_x",&gravity_x);
55      nv_getint("Info.gravity_y",&gravity_y);

56      init_balls();
57      refresh_screen();
58      timer_start(100,"draw_telemetry");
59  }

60  void draw_screen(void) {
61      gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
62      gfx_clearscr(RGB(0,0,0));
63      draw_grid();
64      gfx_setgraphicscontext(GFX_SCREENBUFFER);
65      gfx_copybackground();

66      /* make the buttons */
67      makebutton1("GO!",10,510,100,50,RGB(0,255,0),"go_button");
68      setbuttonfont("GO!", "impact.ttf", 30);

69      makebutton1("STOP!",10,570,100,30,RGB(255,0,0),"stop_button");
70      makebutton1("Reset",120,510,50,90,RGB(255,255,0),"reset_button");
71      makebutton1("Remember",180,510,100,40,RGB(100,0,255),
            "remember_button");
72      makebutton1("Recall",180,560,100,40,RGB(255,0,255),
            "recall_button");
73      makebutton1("More Balls",290,510,100,40,RGB(255,200,200),
            "more_button");
74      makebutton1("Less Balls",290,560,100,40,RGB(200,200,255),
            "less_button");
75      makebutton1("Gravity",400,510,100,40,RGB(200,255,200),
            "gravity_button");
76  }
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-2: example6.c (sheet 3 of 7)

```
77  void draw_grid(void){
78      int x,y;
79      int color;

80      color = BASE_COLOR;
81      for(y=0;y<=SCREEN_HEIGHT-50;y+=50){
82          for(x=0;x<=SCREEN_WIDTH-50;x+=50){
83              gfx_drawbar(x,y,50,50,color);
84              color = ROT_COLOR(color);
85          }
86      }
87  }

88  /** button callbacks **/

89  void go_button(void){
90      sound_play("button.wav");
91      if(!is_moving)
92          timer_start(100,"anim_balls");
93      is_moving = 1;
94  }

95  void stop_button(char * name){
96      sys_debug(">>>> stop button: %s",name);
97      timer_kill("anim_balls");
98      is_moving = 0;
99      sound_play("button.wav");
100 }

101 void reset_button(void){
102     init_balls();
103     refresh_screen();
104     sound_play("button.wav");
105 }

106 void remember_button(void){
107     int i;

108     nv_setchar("Info.initialized",1);
109     nv_setint("Info.num_balls",num_balls);
110     nv_setint("Info.step",step);
111     nv_setint("Info.moving",is_moving);
112     for(i=0;i<num_balls;i++){
113         nv_setint("Ball[%d].x",balls[i].x,i);
114         nv_setint("Ball[%d].y",balls[i].y,i);
115         nv_setint("Ball[%d].x_vel",balls[i].x_vel,i);
116         nv_setint("Ball[%d].y_vel",balls[i].y_vel,i);
117         nv_setint("Ball[%d].handler",balls[i].handler,i);
118     }
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-2: example6.c (sheet 4 of 7)

```
119     sound_play("button.wav");
120 }

121 void recall_button(void){
122     char init;
123     int i;
124     int was_moving;

125     nv_getchar("Info.initialized",&init);
126     if(init){
127         was_moving = is_moving; /*remember moving state for later*/
128         nv_getint("Info.num_balls",&num_balls);
129         nv_getint("Info.step",&step);
130         nv_getint("Info.moving",&is_moving);
131         for(i=0;i<num_balls;i++){
132             nv_getint("Ball[%d].x",&balls[i].x,i);
133             nv_getint("Ball[%d].y",&balls[i].y,i);
134             nv_getint("Ball[%d].x_vel",&balls[i].x_vel,i);
135             nv_getint("Ball[%d].y_vel",&balls[i].y_vel,i);
136             nv_getint("Ball[%d].handler",&balls[i].handler,i);
137         }
138         refresh_screen();
139         if(is_moving && !was_moving)
140             /* this will start a timer only if one is not already */
141             /* going. if one is already going and it needs to be */
142             /* turned off, it will turn itself off after looking at */
143             /* is_moving */
144             timer_start(100,"anim_balls");
145     }
146     sound_play("button.wav");
147 }

148 void more_button(void){
149     if(num_balls < MAX_BALLS){
150         num_balls++;
151         init_ball(num_balls);
152         if(!is_moving)
153             refresh_screen();
154     }
155     sound_play("button.wav");
156 }

157 void less_button(void){
158     if(num_balls>1){
159         num_balls--;
160         if(!is_moving)
161             refresh_screen();
162     }
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-2: example6.c (sheet 5 of 7)

```
163     sound_play("button.wav");
164 }

165 void gravity_button(void){
166     mod_load("gravity");
167 }

168 /** telemetry display **/

169 void draw_telemetry(void){
170     char buf[60];

171     text_printf(buf,"step %d, num. balls %d",step,num_balls);
172     gfx_setfont("georgia.ttf",12);
173     gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
174     gfx_drawbar(500,510,300,30, RGB(0,0,0));
175     gfx_drawstring(510,530,buf,RGB(255,255,255),-1);
176     gfx_setgraphicscontext(GFX_SCREENBUFFER);
177
    gfx_copybufferpart(500,510,300,30,500,510,GFX_BACKGROUNDBUFFER,GFX_SC
    REENBUFFER);

178     timer_start(900,"draw_telemetry");
179 }

180 /** ball animation **/

181 void init_ball(int number){
182     balls[number].x = rnd_get_number(SCREEN_WIDTH - BALL_WIDTH);
183     balls[number].y = rnd_get_number(SCREEN_HEIGHT - BALL_HEIGHT);
184     balls[number].x_vel = rnd_get_number(100) - 50;
185     balls[number].y_vel = rnd_get_number(100) - 50;
186     balls[number].handler = rnd_get_number(NUM_HANDLERS);
187 }

188 void init_balls(void) {
189     int i;
190     for(i=0;i<num_balls;i++){
191         init_ball(i);
192     }
193 }

194 /** animation loop **/

195 void anim_balls(void) {
196     int i;
197     if(!is_moving)
198         return;
199     for(i=0;i<num_balls;i++){
200         switch(balls[i].handler){
201         case 1:
202             calc_pos_rnd(&balls[i]);
```

File Listing 21-2: example6.c (sheet 6 of 7)

```
203             break;
204         default:
205             calc_pos(&balls[i]);
206         }
207     }
208     step++;
209     refresh_screen();
210     timer_start(100,"anim_balls");
211 }

212 /** screen handling functions **/

213 void draw_balls(void) {
214     int i;
215     char ** icon;

216     for(i=0;i<num_balls;i++){
217         if(balls[i].handler == 1)
218             icon = (char**)ball_red;
219         else
220             icon = (char**)ball;  ***NOTE-FIXING -- balls  get stuck
221         gfx_drawsprite(balls[i].x, balls[i].y, icon);
222     }
223 }

224 void refresh_screen(void) {
225     gfx_setgraphicscontext(GFX_BACKGROUNDBUFFER);
226     draw_grid();
227     draw_balls();
228     gfx_setgraphicscontext(GFX_SCREENBUFFER);
229
    gfx_copybufferpart(0,0,SCREEN_WIDTH,SCREEN_HEIGHT,0,0,GFX_BACKGROUNDB
    UFFER,GFX_SCREENBUFFER);
230 }

231 /** ball movement routines **/

232 void calc_pos(ball_struct * b) {
233     b->x += b->x_vel;
234     b->y += b->y_vel;
235     b->x_vel += gravity_x;
236     b->y_vel += gravity_y;

237     if(b->x < 5){
238         b->x = 5;
239         b->x_vel = -(b->x_vel*8/10);
240     }else if(b->x > SCREEN_WIDTH - BALL_WIDTH - 5 ){
241         b->x = SCREEN_WIDTH - BALL_WIDTH - 5;
242         b->x_vel = -(b->x_vel*8/10);
243     }
244     if(b->y < 5){
245         b->y = 5;
246         b->y_vel = -(b->y_vel*8/10);
```

File Listing 21-2: example6.c (sheet 7 of 7)

```
247     }else if(b->y > SCREEN_HEIGHT - BALL_HEIGHT - 5){
248         b->y = SCREEN_HEIGHT - BALL_HEIGHT - 5;
249         b->y_vel = -(b->y_vel*8/10);
250     }
251 }

252 void calc_pos_rnd(ball_struct * b) {
253     b->x += b->x_vel;
254     b->y += b->y_vel;
255     b->x_vel += rnd_get_number(3) - 1;
256     b->y_vel += rnd_get_number(3) - 1;

257     if(b->x < 5){
258         b->x = 5;
259         b->x_vel = -(b->x_vel*8/10);
260     }else if(b->x > SCREEN_WIDTH - BALL_WIDTH - 5 ){
261         b->x = SCREEN_WIDTH - BALL_WIDTH - 5;
262         b->x_vel = -(b->x_vel*8/10);
263     }
264     if(b->y < 5){
265         b->y = 5;
266         b->y_vel = -(b->y_vel*8/10);
267     }else if(b->y > SCREEN_HEIGHT - BALL_HEIGHT - 5){
268         b->y = SCREEN_HEIGHT - BALL_HEIGHT - 5;
269         b->y_vel = -(b->y_vel*8/10);
270     }
271 }
```

This file is included in the SGOS installation CD-ROM tutorial file.

tons to increase and decrease the number of balls.

You also have a new property called **NUM_HANDLERS** with a default value of **2**. "Handlers" refers to the functions that update a ball's position on the screen. This integer value is used in line 186 in **rnd_get_number(NUM_HANDLERS)**. The program randomly chooses one of the handlers and applies it in several routines using **balls[number].handler**.

Ln 19-40 The list of prototypes has grown to cover new buttons and routines.

Ln 42-47 The global variables include the array **ball_struct balls[MAX_BALLS]** to hold all of the ball information, plus **step** to count the animation steps, **num_balls** to track the total number of balls on the screen, and the flag **is_moving** to tell whether animation is active.

Ln 49-59 The **initialize()** function is mostly familiar. There are now more global variables to initialize. The call to **init_balls()** initializes the ball array. Then the call to **refresh_screen()** draws the balls to the screen, and the last line starts a timer for the telemetry display.

Ln 60-76 **draw_screen()** sets the graphics context to the frame buffer, draws the main screen, and copies the frame buffer to the main screen. You then make all seven buttons.

Ln 77-100 The **draw_grid()**, **go_button()**, and **stop_button()** functions are straightforward.

Note that **go_button** checks the animation flag and **stop_button** resets it to **0**.

Ln 101-05 **reset_button()** is simpler since all the drawing of the balls now resides in the subroutine **refresh_screen()**. Now it simply resets all the balls, then redraws the screen.

Ln 106-20 **remember_button()** is a new routine:

The **nv_setchar()** and **nv_setint()** API functions save all the relevant ball information to nvram. Note that if you use a wrong character type, file **debug.warn** will print a warning.

Using **nv_setint("info.num_balls",num_balls)** as an example, the first argument is the name of the variable you are writing to, as a string. The second argument is the value you are saving.

Additional values are substituted into the initial string according to **printf()** style formatting. Any **printf()** formatting characters are allowed, since SGOS makes the substitution using a call to **vsprintf()**. This parameter substitution does not care about syntactic meanings; the string is only interpreted after the substitution. Strings such as **Ball[%d].%s[%d]** or even **%s[%d]** are allowed, as long as they expand into a meaningful string.

Ln 121-47 **recall_button()** now fetches what **remember_button()** stored in nvram. First you confirm that the nvram contains a valid saved position with **info.initialized**. If there is a valid state, you use the **nv_getint()** API function to retrieve the number of balls, the step, and whether the balls were moving.

You then load in all the ball data. If the balls were moving you start a timer to animate the balls, if an animate timer is not already running.

Ln 148-56 **more_button()** adds another ball by incrementing **num_balls**, if the total is still within **MAX_BALLS**. The new ball is initialized the same as the other balls. If the balls are not currently animated, you redraw the screen. Otherwise the screen will be redrawn in the next animation loop.

Ln 157-62 **less_button()** removes a ball by decreasing **num_balls**, to a value as low as **1**. Again, if the balls are not moving you redraw the screen.

Ln 165-67 **gravity_button()** loads the library **gravity**. After compiling, the full name of the file loaded is **gravity.so**. You pass the name with the extension **.so**. As noted at the beginning of this chapter, SGOS changes modules by unloading everything in the current module, calling **mod_exit()**, loading the new module into memory, and calling **initialize()**.

Ln 169-79 **draw_telemetry()** makes a local screen update as discussed in *Graphics Handling Alternatives* at the beginning of this chapter. The routine formats the telemetry string and loads the font you specify, switches to the frame buffer, redraws the background (solid black in this case) with a call to **gfx_drawbar()**, draws the telemetry string with **gfx_drawstring()**, and then copies the boxed area to the screen.

Ln 181-87 **init_ball()** initializes a new ball record. This new function is called by **more_button**, to initialize each added ball. It also randomly assigns a handler to each ball with **balls[number].handler = rnd_get_number(NUM_HANDLERS)**.

Ln 188-93 **init_balls()** re-initializes all the active balls.

L 195-211 **anim_balls()** updates all the ball positions if the flag **is_moving** confirms the pro-

gram is currently animating. Note that each ball is "handled" according to its assigned handler, so balls can have different behaviors.

anim_balls() then redraws the entire screen (in the frame buffer) with a call to **refresh_screen()**. As noted earlier, you simply redraw the entire screen because it has a simple background that redraws quickly.

The last step starts a timer with a callback to **anim_balls** to keep the animation going.

Ln 213-23 **draw_balls()** is familiar except that it now determines which graphic to draw based on the handler for each ball.

Ln 224-30 **refresh_screen()** updates the ball area of the screen by setting the current buffer to the frame buffer, redrawing the colored grid, redrawing the balls and then copying the frame buffer directly to the screen.

Ln 232-51 **calc_pos()** is familiar except that you now tie gravity to a vector.

Ln 252-71 **calc_pos_rnd()** acts as the new ball "handler." It works like **calc_pos()** except that the acceleration on the ball is random.

G. Create Separate Module for gravity.c

File Listing 21-3 shows the code for **gravity.c**, which will be a separate module when the program is compiled. Everything in **gravity.c** should look familiar to or consistent with the rest of the tutorial. The four gravity buttons drive this module and the **back** button exits back to the main module. A few additional notes:

Note that the screen is updated after a button press. Rather than the button redrawing the screen after it changed a value, this tutorial uses an nvram callback to do the update. Looking back in the **game.state** file (*File Listing 21-1*) you will see that both **gravity_x** and **gravity_y** include a callback to **update_gravity()**.

When the main module changes the value of **gravity_x** or **gravity_y** in nvram, it still requests the function **update_gravity**, but will not find it since **update_gravity** is in a different module. The call is harmlessly ignored and the program proceeds without it.

H. Make and Compile

By now the Makefile should look quite familiar. (See File Listing 21-4). Note that you include the new **gravity.c** file with the instruction in line 12:

```
all: game.so gravity.so
```

I. Run the Program

Make and run the program.

After the splash screen, you will see a display similar to the previous example, as shown in *Figure 21-1*. There are four new buttons, **Remember**, **Recall**, **More Balls**, **Less Balls**, and **Gravity**. They respond as follows:

- Pressing **More Balls** adds more balls to the animation loop.

File Listing 21-3: gravity.c (sheet 1 of 3)

```
1   #include "userapi.h"

2   #define SCREEN_WIDTH800
3   #define SCREEN_HEIGHT500

4   #define GRAVITY_X350
5   #define GRAVITY_Y250
6   #define GRAVITY_HW100
7   #define MAX_GRAVITY5
8   #define SCALE_FACTOR((GRAVITY_HW-1)/(2*MAX_GRAVITY))
9   #define CENTER_X(GRAVITY_X + GRAVITY_HW/2)
10  #define CENTER_Y(GRAVITY_Y + GRAVITY_HW/2)

11  void initialize(void);
12  void draw_screen(void);
13  void up_button(void);
14  void down_button(void);
15  void left_button(void);
16  void right_button(void);
17  void update_gravity(void);
18  void draw_gravity(void);
19  void draw_vector(void);
20  double my_sqrt(double a);

21  void initialize(void){
22      draw_screen();
23  }

24  void draw_screen(void){
25      gfx_clearscr(0);
26      gfx_setfont("georgia.ttf",48);
27      gfx_justifystring(0,0,SCREEN_WIDTH,100,"Change Gravity",
28              RGB(200,100,100),JUSTIFY_CENTER,-1);
29      draw_gravity();
30      draw_vector();

31      makebutton1("^",CENTER_X-20,GRAVITY_Y-50,40,40,
32              RGB(100,200,100),"up_button");
33      makebutton1("v",CENTER_X-20,GRAVITY_Y+GRAVITY_HW+10,40,40,
34              RGB(100,200,100),"down_button");
35      makebutton1("<",GRAVITY_X-50,CENTER_Y-20,40,40,
36              RGB(100,200,100),"left_button");
37      makebutton1(">",GRAVITY_X+GRAVITY_HW+10,CENTER_Y-20,40,40,
38              RGB(100,200,100),"right_button");
39      makebutton1("BACK",350,500,100,50,RGB(200,200,200),"back_button");
40  }

41  void up_button(void){
42      int y;
43      nv_getint("Info.gravity_y",&y);
44      y--;
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-3: gravity.c (sheet 2 of 3)

```
45      if(y<-MAX_GRAVITY)
46          y = -MAX_GRAVITY;
47      nv_setint("Info.gravity_y",y);
48      /* update_gravity(); */
49  }

50  void down_button(void){
51      int y;
52      nv_getint("Info.gravity_y",&y);
53      y++;
54      if(y>MAX_GRAVITY)
55          y = MAX_GRAVITY;
56      nv_setint("Info.gravity_y",y);
57      /* update_gravity(); */
58  }

59  void left_button(void){
60      int x;
61      nv_getint("Info.gravity_x",&x);
62      x--;
63      if(x<-MAX_GRAVITY)
64          x = -MAX_GRAVITY;
65      nv_setint("Info.gravity_x",x);
66      /* update_gravity(); */
67  }

68  void right_button(void){
69      int x;
70      nv_getint("Info.gravity_x",&x);
71      x++;
72      if(x>MAX_GRAVITY)
73          x = MAX_GRAVITY;
74      nv_setint("Info.gravity_x",x);
75      /* update_gravity(); */
76  }

77  void back_button(void){
78      mod_exit();
79  }

80  /* graphic routines */

81  void update_gravity(void){
82      draw_gravity();
83      draw_vector();
84  }

85  void draw_vector(void){
86      int g_x,g_y;
87      char buf[60];
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-3: gravity.c (sheet 3 of 3)

```
88      nv_getint("Info.gravity_x",&g_x);
89      nv_getint("Info.gravity_y",&g_y);
90      text_printf(buf,"vec = (%d,%d), |vec| = %g",g_x,g_y,
91            my_sqrt(g_x*g_x+g_y*g_y));
92      gfx_setfont("georgia.ttf",12);
93      gfx_drawbar(0,100,SCREEN_WIDTH,20,RGB(0,0,0));
94      gfx_justifystring(0,100,SCREEN_WIDTH,20,buf,RGB(255,0,0),
95            JUSTIFY_CENTER,-1);
96  }
97  factors  ***** what is this? ******
98  void draw_gravity(void){
99      int center_x,center_y;
100     int g_x,g_y;

101     center_x = CENTER_X;
102     center_y = CENTER_Y;
103     nv_getint("Info.gravity_x",&g_x);
104     nv_getint("Info.gravity_y",&g_y);

105     g_x = center_x + SCALE_FACTOR*g_x;
106     g_y = center_y + SCALE_FACTOR*g_y;

107     gfx_draw3dbar(GRAVITY_X,GRAVITY_Y,GRAVITY_HW,GRAVITY_HW,
108           0,RGB(200,200,255),RGB(200,200,255),2);
109     gfx_drawbar(center_x-5,center_y-5,10,10,RGB(255,0,0));
110     gfx_drawline(center_x,center_y,g_x,g_y,RGB(255,0,0));
111 }

112 /* utility function */

113 double my_sqrt(double a){
114     /* uses newton's method to find the root of the  equation x^2 - a.
    The roots of this equation  are +sqrt(a) and -sqrt(a).
    f(s[n])  s[n+1] = s[n] - ----------                    f'(s[n])
115      */
116     double s;
117     int n;

118     if(a<=0)
119         /* square root of a negative number? */
120         /* also, the square root of zero is zero... */
121         return 0;
122     s = 1;
123     for(n=0;n<10;n++){/* 10 loops should be enough */
124         s = s - (s*s-a)/(2*s);
125     }
126     if(s<0)
127         s = -s;
128     return s;
129 }
130
```

This file is included in the SGOS installation CD-ROM tutorial file.

File Listing 21-4: Makefile file for example6.c

```
1   # Makefile for the examples
2   .SUFFIXES: .cpp .c .so
3   DEBUG=-g
4   SGOS=${shell ./locate sgos}
5   INCLUDE=-I.. -I$(SGOS)

6   .c.o:
7       gcc $(INCLUDE) -c -Wall -fPIC -o $@ $<

8   .cpp.o:
9       g++ $(INCLUDE) $(DEBUG) -c -o $@ $<

10  .o.so:
11      gcc -shared -o $@ $<

12  all: game.so gravity.so

13  game.so: example6.o ball_gfx.o
14      gcc -shared -o game.so example6.o ball_gfx.o
```

This file is included in the SGOS installation CD-ROM tutorial file.

- Pressing **Less Balls** removes balls.
- Pressing **Remember** will store the location, velocity and "personality" of each ball at that moment.
- Pressing **Recall** will restore all the balls to their positions and velocities at the time **Remember** was last pressed. This uses nvram to store the data.

Try pressing **Remember**, then quit, restart, and then press **Recall**. Everything should look as it did before you quit. **Gravity** will load the other module (See *Figure 21-2*), allowing you to change the gravity vector.

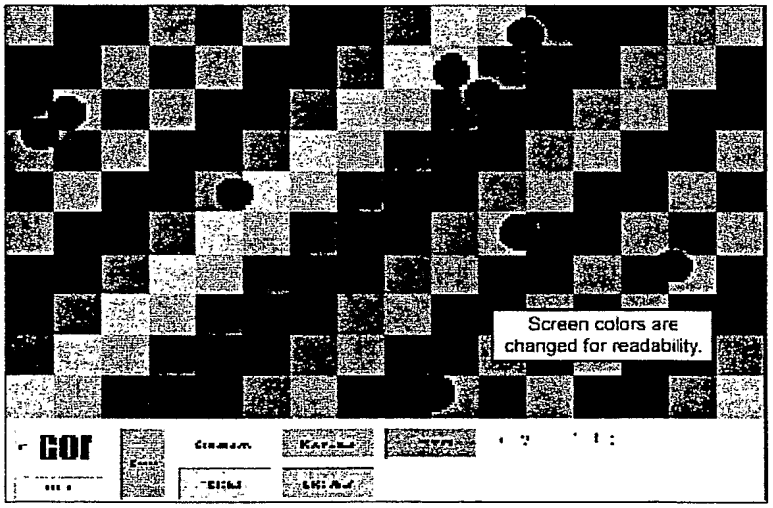


Figure 21-1 – Tutorial: nvram and More Buttons

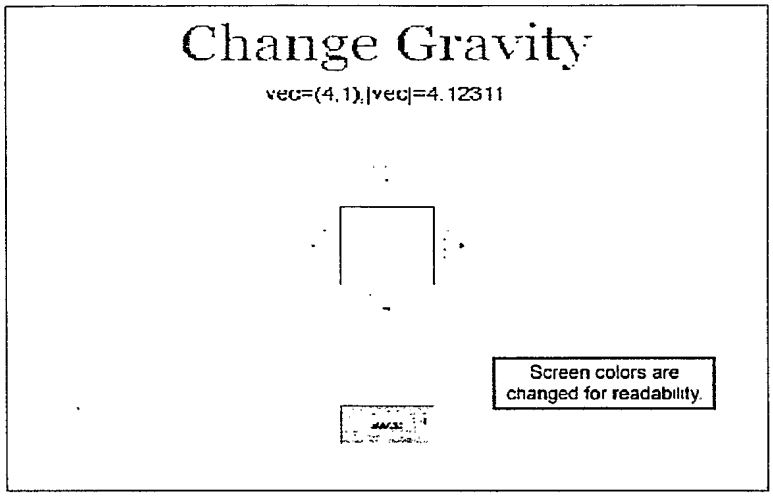


Figure 21-2 – Tutorial: Screen for Second Module

J. EXERCISES

Try the following exercises:

1. Check out the additional tutorial file named **example6b.c** that implements double buffering. The animation loop in this example is extremely flexible. You can probably already see ways to change it. To see how this example would run without double buffering, comment out the 1st, 4th, and 5th lines of **refresh_screen()** so that you draw directly to the screen all the time. Notice the flashing.
2. Also try these adjustments:
   - Make the saved ball position appear when the game is started.
   - Make nvram continually store the ball positions.
   - Create other ball "handlers."
3. Make another module to modify some other parameter, e.g. background pattern or the randomness of the **call_pos_rnd()**. You will need to add some nvram variables to do this. After you change the **game.state** file, delete the nvram file to force SGOS to reinitialize the nvram. You then need to trap on this and initialize the nvram yourself.

V. GAME ENGINE TUTORIAL

Chapter 22 – Build a Simple Game22-1

Chapter 23 – Build a 9-Line Game

CHAPTER 22 — BUILD A SIMPLE GAME

[Put example using generic template here.....]

CHAPTER 23 — BUILD A 9-LINE GAME

A. USE THE 9-LINE GAME TEMPLATE

When available

VI. HARDWARE SOLUTIONS

CHAPTER 24 — HARDWARE SOLUTIONS

A. GAME MAIN MODULE

The tutorials in *IV. API TUTORIAL* include several examples of using the userapi calls.

In some cases the Linux and SGOS software may be preinstalled on hardware provided by Shuffle Master. You may be using PC/104 or other hardware specified by Shuffle Master for a single install to a target machine.

A hardware solution currently available from Shuffle Master is as follows:

[add details.....]

1. URGENT Platform
2. CHIMP – main module
3. HABIT –interface between hardware (lamps, switches, etc.) and computer; includes four serial ports
4. PC/104 Electronics Stack, includes:

[2] a. HIC PC/104 Module – (if faulty, lights and buttons will not work, or game may not come up)
b. Static RAM PC/104 Module – (if faulty, get static RAM error on screen)
c. Operating System PC/104 Module, includes:
M1 – M4 are operating system – (if faulty get no operation)
M6 is the game personality module – if bad get no game or security violation

Newer systems will have a different PC/104 stack, with just two consolidated components. The new HIC module will add SRAM functionality, eliminating the second module. Shuffle Master will provide configuration and installation instructions.

Contact Shuffle Master for currently available and recommended main module hardware.

B. OTHER SUPPORTED HARDWARE

SGOS includes drivers for the following commonly used hardware. Refer to vendor documentation for configuration and installation.

1. Touch Screen Driver

Microtouch

2. Bill Validator Driver

JCM serial and pulse types (or other manufacturers with same protocols)

3. Coin Head Driver

Asai Seiko CC46

4. Coin Hopper Driver

Asai Seiko Hopper Driver

5. Serial Protocols

a. **IGT SAS 402**

a. **Bally SDS**

6. Additional Drivers

Contact Shuffle Master for drivers for other hardware solutions.

C. MECHANICAL REELS

retrofits or new

CHAPTER 25 – ONLINE GAMING ARCHITECTURE (OLGA)

A. NETWORKING PROTOCOLS

Refer to Appendix .....

APPENDIXES

Appendix A – Linux Setup Considerations
Appendix B – make and the Makefile
Appendix C – Embedded userapi Calls
Appendix D – .oti Configuration File
Appendix E – [yourgame].state File
Appendix F – Generic Game Template File
Appendix G – Graphics Conversion Tool Listing
Appendix H – Makestrips Utility (9 Line Games)
Appendix I – Nine Line Game Template
Appendix J – Poker Game Template
Appendix K – Other Templates
Appendix L – Online Protocol Exception Codes
Appendix M – Screens for Setup and Recordkeeping
Appendix N – Advantec Hardware Solution Information
Appendix O – Further Help and Troubleshooting

APPENDIX A – LINUX SETUP CONSIDERATIONS

1. GENERAL NOTES

Although C programming can be done in Windows and then copied to Linux, it will be simplest to work directly with SGOS in Linux. You can install Linux most easily on a desktop PC, but a laptop is also an option. Laptops often use specialized hardware, so finding proper Linux drivers can be more difficult.

The steps to set up SGOS on the development computer depend on your exact situation:

2. IF LINUX IS ALREADY LOADED

If you already have Red Hat Linux 6.x loaded on your computer, you can go ahead and install SGOS from the provided CD-ROM .

3. INSTALLING LINUX ON A DEDICATED COMPUTER

Linux can function well on an older Pentium computer with 32MB of RAM (64 is better). A dedicated computer will simplify the installation.

4. SHARING WITH A WINDOWS COMPUTER

To install Red Hat Linux alongside Windows on an existing computer requires a dedicated hard drive or a dedicated drive partition.

Once again Linux requirements are economical. To keep things simple, install a separate drive if possible. An outgrown 2 or 3 GB drive from a Windows computer will be quite sufficient for game development with SGOS.

5. USING THE SHUFFLE MASTER DEVELOPMENT SYSTEM

Shuffle Master offers a turnkey hardware and software unit for development with SGOS. These units ship with the appropriate software loaded.

6. ADDITIONAL HELP

If you are a novice with Linux — and especially if you will have a dual boot system with both Windows and Linux — pick up one of the many Red Hat Linux books available. For example, *Red Hat Linux for Dummies* gives a good explanation of dual-boot systems and how to resolve common problems. You will also find abundant help on the internet. Three helpful sites are linuxcare.com, support.com, and questionsexchange.com.

APPENDIX B – MAKE AND THE MAKEFILE

1. OVERVIEW

make is a Linux tool to organize, update, compile and link the files that make up your program. When you run make it looks first for a file called **makefile** to tell it about the dependencies in your program files. If it does not find **makefile**, it next looks for a **Makefile**. SGOS is set up to use the one with an upper case "M", so it stands out in the listings. The **Makefile** is basically a set of rules to tell **make** precisely how it should construct non-source files from other files so it can build and install the entire program.

Each time you run **make** (by typing **make** at the command line), it will check which source files have changed and update every affected file in your project. Once it is set up correctly, the **Makefile** fully maintains your files. Any time you change source code, invoking **make** will immediately rebuild your project.

SGOS includes a **Makefile** for each game template and for each tutorial. The tutorial Makefiles are more simple than the example which follows, but every Makefile includes the same basic steps. If you are new to Linux (**Makefile** is originally a Unix tool, adapted to Linux), it is worth spending some time getting to know the Makefile structure and rules. This manual can get you started with the **make** tool as it is used to build games with SGOS templates. The **make** tool and the **Makefile** offer many very complex and powerful options. Picking up a good Linux manual is highly recommended.

2. A MAKEFILE EXAMPLE

A breakdown of the Makefile for the Press Your Luck 9-line game (File Listing C-1) is as follows:

Line 2: **.SUFFIXES: .cpp .c .so** tells make the list of known suffixes for files in the directory that it needs to use in this makefile.

Line 3: **DEBUG=-g** is used as a compiler flag. It tells the compiler to generate debugging symbols. See Chapter XXX for more on debugging your game in the SGOS.

Ln 4-8: The following lines define some variables **SHARED, ENGINE, SGOS, LIBS and INCLUDE**, find the path to **userapi.h** and all of its associated SGOS libraries and include the path as a string command line option to the compiler.

```
SHARED=${shell ./locate shared}
ENGINE=${shell ./locate engine}
SGOS=${shell ./locate sgos}
LIBS=-L$(SHARED)
INCLUDE=-I.. -I$(SGOS) -I$(SHARED) -I$(ENGINE)
```

Line 9: **SRCS=$(shell ls *.c)** finds all the **.c** files n the directory and assigns them to the variable **SRCS**.

Line 10: **TARGETS=$(SRCS:.c=.so)** changes the suffix of all the **.c** files in **SRCS** to **.so**. and assigns them to the variable **TARGETS**.

File Listing B-1: sample Makefile listing

```
1   # Makefile for nineline
2   .SUFFIXES: .cpp .c .so

3   DEBUG=-g
4   SHARED=${shell ./locate shared}
5   ENGINE=${shell ./locate engine}
6   SGOS=${shell ./locate sgos}
7   LIBS=-L$(SHARED)
8   INCLUDE=-I.. -I$(SGOS) -I$(SHARED) -I$(ENGINE)
9   SRCS=$(shell ls *.c)
10      TARGETS=$(SRCS:.c=.so)
11      GAMEOBJECTS=nineline.o pyl_gfx.o

12  .c.o:
13      gcc $(INCLUDE) -c -Wall -fPIC -o $@ $<

14  .cpp.o:
15      g++ $(INCLUDE) $(DEBUG) -c -o $@ $<

16  .o.so:
17      gcc -shared -o $@ $<

18  all: game.so $(TARGETS) cleanup

19  game.so: $(GAMEOBJECTS)
20      gcc -shared -o game.so $(GAMEOBJECTS) $(LIBS) -lengine -lnineline

21  last_games.so: last_games.o lastgame_gfx.o
22      gcc -shared -o last_games.so last_games.o lastgame_gfx.o $(LIBS) -llast-
    games -llastnine

23  bonus.so: bonus.o bonus_gfx.o
24      gcc -shared -o bonus.so bonus.o bonus_gfx.o

25  paytable.so: paytable.o pay_gfx.o
26      gcc -shared -o paytable.so paytable.o pay_gfx.o

27  cleanup:
28      rm -f nineline.so debug*

29  clean:
30      rm -f *.o *.so nvram debug.* core

31  sos:
32      @echo "Enter ROOT password to build app image..."
33      @su -c ./makesos

34  # Game Dependencies
35  bonus.o: bonus.c bonus_gfx.h
36  game_books.o: game_books.c
37  help.o: help.c
38  last_games.o: last_games.c lastgame_gfx.h nineline.h
39  nineline.o: nineline.c pyl_gfx.h nineline.h
40  paytable.o: paytable.c pay_gfx.h nineline.h
```

This sample listing is from the Press Your Luck nine-line game.

Line 11: **GAMEOBJECTS=nineline.o pyl_gfx.o** associates the files **nineline.o** and **pyl_gfx.o** to the variable **GAMEOBJECTS.**

Line 12: **.c.o:** is a generic rule for creating **.o** files from **.c** files.

Line 13: **gcc $(INCLUDE) -c -Wall -fPIC -o $@ $<** defines the generic rule, as follows:

This line must begin with a tab to be recognized by make as a command by the **gcc C** compiler.

$(INCLUDE) provides the path to userapi.h found above.

-c compiles each source file to an object file, but does not link.

-Wall warns about all questionable constructs.

-fPIC turns on position independent code. This option is necessary.

-o $@** places the output in the file **$@ (shorthand for the name of the target, expands to **file.o**).

$< is shorthand for the name of the input file (expands to **file.c**).

Ln 14-17: **.cpp.o:**

g++ $(INCLUDE) $(DEBUG) -c -o $@ $<

.o.so:

gcc -shared -o $@ $<

.cpp.o and **.o.so** give general rules for turning **C++** files into object files, and for turning object files in shared object files.

Line 18: **all: game.so $(TARGETS) cleanup** states that to make everything, make must create **game.so** from all the **.so**'s in **TARGETS** and execute the **cleanup**.

Line 19: **game.so: $(GAMEOBJECTS)** states that **game.so** is dependent on **GAMEOBJECTS**, which expands into a list of files.

Line 20: **gcc -shared -o game.so $(GAMEOBJECTS) $(LIBS) -lengine -lninel ine** makes a shared object out of **GAMEOBJECTS**, **LIBS**, links in the **engine** and **nineline** libraries, and names it **game.so.**

Ln 21-22: The next two lines state that **last_games.so** is dependent on **last_games.o** and **lastgame_gfx.o**. It makes the shared object **last_games.so.**

last_games.so: last_games.o lastgame_gfx.o

gcc -shared -o last_games.so last_games.o lastgame_gfx.o $(LIBS) -llast-games -llastnine

Ln 23-24: The following lines state that **bonus.so** is dependent on **bonus.o** and **bonus_gfx.o.** They make the shared object **bonus.so.**

bonus.so: bonus.o bonus_gfx.o

gcc -shared -o bonus.so bonus.o bonus_gfx.o

The following lines state that **paytable.so** is dependent on **paytable.o** and **pay_gfx.o.** It makes the shared object **paytable.so.**

Ln 25-26: **paytable.so: paytable.o pay_gfx.o**

gcc -shared -o paytable.so paytable.o pay_gfx.o

The next two lines remove the **nineline.so** file and all **debug** files. The **rm** remove command has the **-f** forced option, so there will be no asking for confirmation.

Ln 27-28:
```
cleanup:
rm -f nineline.so debug*
```

Ln 29-30: The following lines act the same as those above but they remove all objects, shared objects, **nvram**, **debug** files and any **core** dump files that may exist.

```
clean:
rm -f *.o *.so nvram debug.* core
```

Ln 31-33: The following lines will start the script that makes the shared objects for the disk on chip. You must be super user, and will be prompted for the password.

```
sos:
@echo "Enter ROOT password to build app image..."
@su -c ./makesos
```

Ln 35-40: The following lines show the associated dependencies for the object files on the **.c** source and **.h** header files.

```
bonus.o: bonus.c bonus_gfx.h
game_books.o: game_books.c
help.o: help.c
last_games.o: last_games.c lastgame_gfx.h nineline.h
nineline.o: nineline.c pyl_gfx.h nineline.h
paytable.o: paytable.c pay_gfx.h nineline.h
```

3. RUNNING MAKE

To create the **game.so** file, type

```
make
```

To use this makefile to delete the executable and object files from the directory, type

```
make clean
```

To rebuild the entire program (recompile and link all objects, not just the time stamped ones) type

```
make all
```

To create **so**'s for use in making the Disk On Chips, (as root) type

```
make sos
```

APPENDIX C – EMBEDDED USERAPI CALLS

1. GENERAL NOTES ABOUT USERAPI CALLS

userapi.h declares functions for graphics, sound, timers, and several other game operations. They are a versatile set of functions for graphics, widgets, module handling, timers, nvram access, sound, hardware interface, text formatting, system calls, multigame manamgment and miscellaneous routines.

The tutorials in *IV. API TUTORIAL* include several examples of using the **userapi** calls.

2. GRAPHICS ROUTINES

The graphic routines have a few conventions for describing rectangles and colors. Rectangles are defined by the 4-tuple **(x, y, w, h)**, where **(x, y)** is the location of the upper left corner and **(w, h)** is the width and height of the rectangle. Colors are specified using integers, which represent an RGB triple. The macro **RGB(r, g, b)** in **userapi.h** converts RGB values to their corresponding integer. R,G,B are in the range 0 and 255.

The API graphics functions can interact with one of the three SGOS buffers.This context gets set by **gfx_setcontext**. Discussion in *Chapter 7* reviews the relevant contexts for each function. They are:

- **GFX_SCREENBUFFER**
- **GFX_BACKGROUNDBUFFER**
- **GFX_WORKBUFFER.**

The transform flags are:

- **GFX_FLIP_HORIZ**
- **GFX_FLIP_VERT**
- **GFX_ROTATE_90**
- **GFX_ROTATE_180**
- **GFX_ROTATE_270**

The transforms are applied in the following (arbitrary) order:

image ⇨ flip horizontal ⇨ flip vertical ⇨ rotates ⇨ result

The text justification flags are:

- **GFX_JUSTIFY_HORIZ_CENTER**
- **GFX_JUSTIFY_HORIZ_LEFT**
- **GFX_JUSTIFY_HORIZ_RIGHT**
- **GFX_JUSTIFY_VERT_CENTER**
- **GFX_JUSTIFY_VERT_TOP**
- **GFX_JUSTIFY_VERT_BOTTOM**
- **GFX_JUSTIFY_CENTER**

gfx_clearbuffer(int color)
Clears the entire buffer with color.

gfx_copybuffer(int x, int y, int w, int h, int destx, int desty, int srcbuffer, int destbuffer)
Copies the box specified by **x,y,w,h** from the source buffer to the destination buffer at **destx, desty.**

gfx_draw3dbar(int x, int y, int w, int h, int fill, int ul_fill, int lr_fill, int bdwidth)
The rectangle is at **x,y,w,h. fill, ul_fill**, and **lr_fill** are colors. The color of the center of the rectangle is **fill**. Passing a **-1** for **fill** indicates a transparent center, allowing you to draw a framed rectangle. **ul_fill** is the color of the upper and left sides of the rectangle; **lr_fill** is the color of the bottom and right sides.

gfx_drawbar(int x, int y, int w, int h, int c)
Draws a solid rectangle at **x,y** with width **w**, height **h**, and color **c**.

gfx_drawicon(int type, int x, int y, void *bitmap, int trans_color, int xoff, int yoff, int w, int h, int transform)
*** add type info etc. when done

Draws the **bitmap** on the active buffer with the upper left corner at **x,y**. xoff and yoff are the offsets in the image of the cropped rectangle. **w,h** are the width and height of the rectangle and transform is the orientation (0, 90, 180, 270). .

TIP... Passing **-1** for both the **x** and **y** loads the entire bitmap but suppresses the drawing. This provides a way to cache an image in memory.

gfx_drawline(int x1, int y1, int x2, int y2, int c)
Draws a line 1 pixel wide from the point **x1,y1** to the point **x2,y2** with color **c**.

???gfx_drawpixel(int x, int y, int c) is this still here do we use drawline call
Draws a pixel at **x,y** the color **c**.

???is this drawsprite or drawbitmap since we are passing in the type, what are the valid types also ...

gfx_drawsprite(int type, int x, int y, void *bitmap, int xoff, int yoff, int w, int h, int transform)
Similar to **gfx_drawicon()**, except bitmap must use color **99** as the transparent color.

int gfx_drawstring(int x, int y, int w, int h, int transform, char* str, int color, int justification, int bgcolor)
Writes a zero-terminated string **str** to the buffer. **x,y** is the lower left corner of the string, transform is 0, 90, 180, or 270, **color** is the text color, **justificaiton** desired, and

bgcolor is the background color. Passing **-1** for **bgcolor** makes the background to be transparent. Note that **gfx_drawstring()** can handle new line characters (\n).

gfx_geticondimensions(void *icon, int *w, int *h)
Returns the w and h of he specified bitmap.

??? is this supposed to be in here???

gfx_getrect(int x, int y, int w, int h, void *buf)
This routine copies the entire rectangle x, y, w, h, of the specified buffer.

gfx_setcachesize(int size)
Sets the current cache size in megabytes used for graphics.

gfx_setcaching(int onoff)
Sets the use of the cache on or off for graphics.

gfx_setclipping(int onoff)
Sets automatic graphics clipping on or off.

gfx_setclippingrect(int x, int y, int w, int h)
Sets the current location and size used for clipping.

gfx_setcontext(int context)
Sets the current graphics context. **context** is one of the following values:

- **GFX_SCREENBUFFER**
- **GFX_BACKGROUNDBUFFER**
- **GFX_WORKBUFFER**

All commands, unless specifically stated, work the same whether drawing to the screen (**GFX_SCREENBUFFER**) or drawing to memory (**GFX_BACKGROUNDBUFFER, GFX_WORKBUFFER**). There is video clipping provided for each buffer.

gfx_setfont(char* fontfile, int size)
Loads the font in the file **fontfile**, and makes it the current font. Sets the current font size to **size**.

3. WIDGET ROUTINES

*** This entire section is changing. We will speicify default atrtributes and then use MACROS to modify those.

SGOS buttons respond to finger presses on a touchscreen. For the development system on your desktop computer you can activate a button with a mouse. Each button calls a callback function when pressed. These callback functions may have one argument which is the name of the button pressed. This allows many buttons to have the same callback function, which is useful when putting keypads, or other similar controls on the screen.

* While there is only really one kind of button, there are three ways to make it. The sim-

plest way to make a button is to give it a color and a name. The name is then displayed on the screen. The second way to make a button is to associate an icon (or graphic) with it; the icon is then displayed on the screen instead of the button name. The third way is to pass two icons, one for the button as it normally is and one for when the button is pressed.

* You can change the parameters of a button by referring to it by name.

```
widget_action(...???)

widget_draw(char *name)

widget_getattribute(char *name, char * attribute, int *value)

widget_make(int type, char *name, int x, int y, int w, int h)

widget_setattribute(char *name, char *attribute, int value)
```

4. MODULE HANDLING ROUTINES

mod_exit(void)
Fetches the library name most recently saved by mod_load(), and does a **mod_ump()** to it. mod_exit() also removes the library name from the saved list.

mod_jump(char *mod_name)
Just like **mod_load()**, except the current library name is not remembered.

mod_load(char *mod_name)
Loads the library **mod_name**, and puts a request for the function **initialize()** in the event queue. **mod_load()** also saves the current library name for retrieval by mod_exit().

5. TIMER ROUTINES

timer_start(long timeout, char* callback)
Starts a timer of duration **timeout** in milliseconds. When the timer expires, a request for the function **callback** is put in the event queue. **callback** is a string giving the function name. The callback function must be passed without arguments.

timer_kill(char* callback)
Deletes all the timers having the callback function **callback**. The function named **callback** is not called.

6. NON-VOLATILE RAM ROUTINES

The nvram is managed by a set of special routines. The game application never directly touches anything in the nvram; all game manipulations of the nvram occur through these func-

tions. The nvram is partitioned into variables by the nvram manager according to the contents of the **mygame.state** file. This text file looks similar to a list of C structure declarations, except the comment character is a # and this line never passes through a parser. The nvram parser reads this file at startup.

The **mygame.state** file makes it easy to put variables in the nvram, since the game application never needs to be concerned with the offsets of variables in the nvram. Structure definitions can even be nested.

An **nvram string** refers to a specific variable in nvram. A typical one looks like **Game.creditsleft**. To allow access to arrays, standard **printf**-style format characters can be put in the string, and the values to insert are supplied as the additional arguments on each line. This feature allows strings like **History[%l].bet, Stats.icon[%l]**, or even **BillHistory[%l].date[%l]**. More elaborate formats are also possible, such as **MyStruct.%s[%l]**, since the string is formatted before being parsed.

The following set of procedures sets an nvram variable to the passed value. Use the variant that matches the data type of the nvram variable. The additional arguments are the **nvramstr** formatting values, if needed.

```
nv_setchar(char* nvramstr, char, ...)

nv_setdouble(char* nvramstr, double, ...)

nv_setfloat(char* nvramstr, float, ...)

nv_setint(char* nvramstr, int, ...)

nv_setlong(char* nvramstr, long, ...)

nv_setshort(char* nvramstr, short, ...)
```

The following functions retrieve a value from an nvram variable. The value is returned through the pointer **val**. The function itself returns nothing. Use the variant matching the data type of the nvram variable. The additional arguments are the **nvramstr** formatting values, if needed.

```
nv_getchar(char* nvramstr, char* val, ...)

nv_getdouble(char* nvramstr, double* val, ...)

nv_getfloat(char* nvramstr, float* val, ...)

nv_getint(char* nvramstr, int* val, ...)

nv_getlong(char* nvramstr, long* val, ...)
```

nv_getshort(char* nvramstr, short* val, ...)

The following routines increment an nvram by **amt**. **char*** may be negative. No values are returned. Use the variant matching the data type of the nvram variable. The additional arguments are the **nvramstr** formatting values, if needed.

nv_incchar(char* nvramstr, char amt, ...)

nv_incdouble(char* nvramstr, double amt, ...)

nv_incfloat(char* nvramstr, float amt, ...)

nv_incint(char* nvramstr, int amt, ...)

nv_inclong(char* nvramstr, long amt, ...)

nv_incshort(char* nvramstr, short amt, ...)

7. SOUND ROUTINES

sound_play(char* file, int channel, int loop, int pan)
Plays the sound file named **char* file**.

- **int loop** is either **0** or **1**. **0** plays once; **1** starts a loop which ends with a call to **sound_stop()**.
- **int channel** sets the sound to one of the 32 available channels.
- **int pan** ranges from **-2** to **2**, where **-2** is left, **2** is right speaker.

sound_stop(int channel)
Stops the sound currently playing in the specified channel, 1 through 32.

sound_volume(int percent)
Sets the sound volume to **percent** percent of maximum volume.

8. MECHANICAL REEL ROUTINES

reel_spin(unsigned char* reelstop, int numstops)
Initiates the spin of the mechanical reels, and stops those reels at array **reelstop**, values 0-21, for the number of reels **numstops**, default value 3.

reels_stop(unsigned char reelmask)
Stops all spinning reels from **reelmask**. Can be used for a skill stop.

9. External Display Routines

extdisp_award(char* AwardChar)
Displays the award to the external display.

extdisp_bet(char* BetChar)
Displays the bet to the external display.

extdisp_credits(char* CreditsChar)
Displays the credits to the external display.

extdisp_get(int type, char* DispString)
Reads the currently displayed text string from the external display of type into **DispString**.

extdisp_set(int type, char* DispString)
Sends the string **DispString** to the external display of type. Supported external display types defined in **userapi.h** are:

- **LED_DISPLAY**

10. Text Formatting routines

int text_printf(char* str, const char* format, ...)
Prints **format**, with any additional variables, into the buffer pointed to by **str**. This function does not allocate any memory for the string.

int text_strcat(char* dest, const char* src)
Concatenates a copy of **src** to **dest**. **src** is untouched by this operation.

int text_strcmp(const char* str1, const char* str2)
Alphabetically compares **str1** and **str2** and returns an integer based on the outcome:

Value	Meaning
Less than zero	**str1** is less that **str2**
Zero	**str1** is equal to **str2**
Greater than zero	**str1** is greater than **str**

11. Resource routines

These resource functions ...

resource_get(char *what, ...)

resource_set_file_opt(char *base_name, int must_exist)

This is usually used through the #defined funtion resource-set_file(base_name).

Notes: Strings returned are sopied into the buffer given, THe strings are guarenteed not to be longer that RESOURCE_MAXSTRLEN bytes long.

12. MISCELLANEOUS ROUTINES

These miscellaneous functions do not fit in any of the above categories.

sys_breakpoint(int tag, void* param)

Since the debugger will not allow breakpoints to be set in the game code (because the game is technically a "library"), calling this function in your game code cleverly allows you to trap breakpoints in the debugger. Passing a pointer in the argument **param** allows you view and alter a game variable in the debugger. The function **sys_breakpoint** is listed below for reference:

```
void sys_breakpoint(int tag, void* param){
        char char_setval = -1;
        short short_setval = -1;
        int int_setval = -1;
        if (char_setval != -1) {
                char* p = (char*)param;
                *p = char_setval;
        }
        if (short_setval != -1) {
                short* p = (short*)param;
                *p = short_setval;
        }
        if (int_setval != -1) {
                int* p = (int*)param;
                *p = int_setval;
        }
}
```

sys_debug(char* format, ...)

Prints **format**, which may contain **printf**-style formatting characters, to the file **debug.out**.

sys_exec(char* func)

Executes the function **func** and returns **NULL** if the function does not exist. **func** is a string giving the function name.

unsigned long rnd_get_number(unsigned long range)

Returns a random number "r" such that 0 <= r < **range**.

sys_setlamp(char *light, int state)

Sets light **light** to **state**. **light** is a string defined in the **.oti** file.

unsigned long sys_clock()

Returns the system clock time.

char* sys_getreleasedate(void)
Returns a string stating the release date of the running version of SGOS.

char* sys_getversion(void)
Returns a string stating the running version of SGOS.

char* sys_getversion(void)
Returns a string stating the running version of SGOS.

13. ENGINE API CALLS

The following API calls become available by including **engine_api.h** in the Makefile:

```
#define TYPE_SOLO       0

#define TYPE_MULTI      1
```

A. Credit Engine Data

engine_changebet(int change)
Change is in (+/-) units of **betinc**

long engine_getaward(void)
Returns the current total amount won.

int engine_getbet(void)
Returns the amount of the current bet.

int engine_getbetinc(void)
Returns the value of the bet incrementor.

long engine_getcredits(void)
Returns the current number of credits on the machine.

char engine_getdoorclosedflag(void)
Returns the status of the door closed flag, **1** closed, **0** open

long engine_gethandpay(void)
Returns the **handpay** amount.

long engine_getpaid(void)
Returns the current amount won.

long engine_getpaidout(void)
Returns the amount collected from hopper.

char engine_getpowerresetflag(void)
Returns the status of the power reset flag: **1** power reset, **0** not.

char engine_getrebet(void)
Returns the **rebet** flag value.

engine_setbetinc(int new_betinc)
Set the new bet incrementor, in credits.

engine_setsnap(void)
Sets a snap for the credit countdown.

B. Managing a Multigame

char game_getcurrentstate(void)
Returns the current game state.

int game_getdenomination(void)
Returns the amount of the current denomination.

int game_getmaxbet(void)
Returns the current **maxbet** amount of the machine

int game_inprogress(void)
Returns status **1** if a game is in progress, or **0** on false.

game_load(char *name)
Sets **currentgame** to active instance of **name** and loads the **lib**

game_register(char *name, int denom, int maxbet, int percent, char type)

game_setcurrentstate(char state)
Sets the game state to **state**.

game_setdenomination(int denom)
Sets **currentgame** to **currentgame name** with **denom**

C. Miscellaneous

char * text_iconvert(int num, int pad)

char * text_lconvert(long num, int pad)

char reel_isstopped(int reelnum)
Returns the state of reel (1...5)

14. GAME SPECIFIC API CALLS

A. Nineline Games

The following API calls become available by including **nineline_api.h** in the Makefile:

char getbetperline(void)
Returns the current bet per line.

char getLinesBet(void)
Returns the number of lines that have bets placed

B. Poker Games

The following API calls become available by including **poker_api.h** in the Makefile (empty for now)

APPENDIX D — .OTI CONFIGURATION FILE

Naming of **.oti** and **.state** files is derived from the base name given in the API call **RegisterGame()**. If your application has multiple games, each game will have its own **.oti** file. Multiple denominations for the same game will share the same **.oti** file.

1. MYGAME.OTI FILE LISTING

You will modify the default **.oti** file for your game's hardware, if different. For instance, an extra button on the target machine will require a related mapping.

Refer to parts B and C of this chapter for more about **.oti** syntax and file addresses.

The following is an example of a typical **.oti** file. Make changes as needed for your game configuration.

```
1   # Shuffle Master Template OTI file
2
3   ####################
4   # Begin OTI Data   #
5   ####################
6   runtime : {
7      load : game;
8   }
9   startup : {
10     display;
11     sound;
12     touchscreen;
13     chimp;
14     billacceptor;
15     protocol;
16  }
17  serial : {
18     1, touchscreen;
19     2, billacceptor;
20     3, protocol;
21     4, watchdog;
22  }
23  debug : {
24     output : file;
25     options : {all;}
26  }
27  nvram : {
28     media : file;
29  }
30  kernel : {
31     "snd-card-es1688.o snd_irq=9";
32  }
33  output : {
34     <3,2>, LOCKOUT;
35     <3,5>, HOPPERMTRL;
```

```
36      <3,6>, HOPPERMTRH;
37      <3,7>, MUTESWITCH;
38      <2,6>, DIVERTERA;
39      <3,0>, DIVERTERB;
40  }
41  portmap : {
42      <P,4,0>, z, main_open;
43      <R,4,0>, a, main_closed;
44      <P,4,1>, c, logic_open;
45      <R,4,1>, d, logic_closed;
46      <P,4,2>, A, hopper_closed;
47      <R,4,2>, Z, hopper_open;
48      <P,4,3>, S, printer_closed;
49      <R,4,3>, X, printer_open;
50      <P,4,4>, s, cash_closed;
51      <R,4,4>, x, cash_open;
52      <P,4,5>, D, drop_closed;
53      <R,4,5>, C, drop_open;
54      <P,3,0>, ',', coindn_switch;
55      <R,3,0>, k, coinup_switch;
56      <P,2,2>, '#', coinrev_switch;
57      <P,3,2>, l, hopperup_switch;
58      <R,3,2>, '.', hopperdn_switch;
59      <P,3,6>, o, hopperfullup_switch;
60      <R,3,6>, p, hopperfulldn_switch;
61      # The, following are synthesized events!
62      <S,0,0>, f, bill_clear;
63      <S,0,1>, v, bill_error;
64      <S,0,3>, b, printer_clear;
65      <S,0,4>, j, billup_switch;
66      <S,0,5>, m, billdn_switch;
67      <S,0,6>, w, bill1_switch;
68      <S,0,7>, e, bill2_switch;
69      <S,1,0>, r, bill5_switch;
70      <S,1,1>, t, bill10_switch;
71      <S,1,2>, y, bill20_switch;
72      <S,1,3>, u, bill50_switch;
73      <S,1,4>, i, bill100_switch;
74  }
75  panel : {
76      <P,2,7>, 1, service_switch,      switch_1;
77      <P,3,1>, 2, collect_switch,      switch_2;
78      <P,0,3>, 3, betoneline_switch,   switch_3;
79      <P,0,5>, 4, betthreeline_switch, switch_4;
80      <P,0,7>, 5, betfiveline_switch,  switch_5;
81      <P,1,1>, 6, betsevenline_switch, switch_6;
82      <P,1,3>, 7, betnineline_switch,  switch_7;
83      <P,1,5>, 8, bet1perline_switch,  switch_8;
84      <P,1,7>, 9, bet2perline_switch,  switch_9;
85      <P,2,1>, 0, bet3perline_switch,  switch_10;
86      <P,2,3>, '-', bet4perline_switch,  switch_11;
87      <P,2,5>, '=', bet5perline_switch,  switch_12;
88      <P,0,1>, ']', spin_switch,         switch_13;
89      <P,0,2>, ';', setup_switch,        switch_14;
```

```
90      <P,0,0>, '/', jackpot_switch,       switch_15;
91   }
92   lights : {
93      #These light strings are used by the engine
94      <0,0>, towera light;
95      <0,2>, towerb light;
96      <0,4>, towerc light;
97      <2,7>, service light;
98      #These light strings are user defined
99      <0,1>, spin light;
100     <0,3>, play1 light;
101     <0,5>, play3 light;
102     <0,7>, play5 light;
103     <1,1>, play7 light;
104     <1,3>, play9 light;
105     <1,5>, bet1 light;
106     <1,7>, bet2 light;
107     <2,1>, bet3 light;
108     <2,3>, bet4 light;
109     <2,5>, bet5 light;
110     <3,1>, collect light;
111  }
112  coordinates : {
113  # These are 'shared' widgets used by the engine
114     credit box : 5,500,70,30;
115     info box : 400,475,400,20;
116     paid box : 270,500,70,30;
117     bet box : 620,500,50,30;
118     payout box : 110,500,125,30;
119     power reset : 10,70,100,10;
120     door closed : 700,70,100,10;
121     win-a : 200,60,400,20;
122     win-b : 180,470,100,20;
123     # Any 'custom' widgets should go here...
124     lines box : 460,500,50,30;
125     bet per line box : 540,500,50,30;
126  }
127  info strings : {
128     GOOD LUCK;
129     GAME OVER;
130     BET 5 CREDITS;
131     PUSH SPIN;
132     INSERT BILL;
133  }
134  ##########################################################
135  # Critical Event Function Pointers                       #
136  #                                                        #
137  # Below is a reference list of the currently supported   #
138  # functions calls in the engine keyhandler...            #
139  #                                                        #
140  # THIS SECTION SHOULD NOT BE CHANGED! #
141  ##########################################################
142  keyhandler : {
143     main_closed;
```

```
144     main_open;
145     cash_closed;
146     cash_open;
147     logic_closed;
148     logic_open;
149     hopper_closed;
150     hopper_open;
151     printer_closed;
152     printer_open;
153     drop_closed;
154     drop_open;
155     bill_clear;
156     bill_error;
157     printer_error;
158     printer_clear;
159     billup_switch;
160     billdn_switch;
161     coinup_switch;
162     coindn_switch;
163     coinrev_switch;
164     hopperup_switch;
165     hopperdn_switch;
166     hopperfullup_switch;
167     hopperfulldn_switch;
168     bill1_switch;
169     bill2_switch;
170     bill5_switch;
171     bill10_switch;
172     bill20_switch;
173     bill50_switch;
174     bill100_switch;
175 }
176 meters : {
177     <4,3>; # Total In
178     <4,6>; # Total Out
179     <4,0>; # Drop
180     <4,5>; # Credits Paid
181     <4,4>; # Jackpot
182     <4,1>; # Progressive
183 }
184
185
```

2. .OTI SYNTAX FOR THE CORE SYSTEM

The following are the syntax rules for **.oti**.

```
1   syntax : group {
2   # the resource manager expects the following definitions
3       serial : array of i=port, s=what / port >= 1 and port <= 4,
4                   what = watchdog
5                   or what = touchscreen
6                   or what = billacceptor
```

```
7                or what = protocol;
8      kernel : array of S;
9      output : array of <i=port,i=bit>, s=what /
10         port >= 0 and port <= 7,
11         bit >= 0 and bit <= 7,
12                  what = lockout or what = hoppermtrl or what = hoppermtrh
13         or what = muteswitch or what = divertera or what = diverterb;
14     portmap : array of <s=porttype,i=port,i=bit>, S=letter, S=function /
15                  porttype = p or porttype = r or porttype = s,
16         port >= 0 and port <= 7,
17         bit >= 0 and bit <= 7,
18         len(letter) = 1;
19     panel : array of <s=porttype,i=port,i=bit>, S=letter, s=cb,s=cb2 /
20                  porttype = p or porttype = r or porttype = s,
21         port >= 0 and port <= 4,
22         bit >= 0 and bit <= 7,
23         len(letter) = 1;
24     lights : array of <i=port,i=bit>, s / port >= 0  and port <= 7,
25         bit >= 0 and bit <=7;
26     meters : array of <i=port, i=bit> / port >= 0 and port <= 7,
27         bit >= 0 and bit <= 7;
28     keyhandler : array of S;
29
30  # the other parts of the system use the following
31     runtime : group {
32        load : S;
33        ? preload : array of S;
34     }
35     startup : array of s=what /
36                     what = display
37                  or what = sound
38                  or what = touchscreen
39                  or what = chimp
40                  or what = billacceptor
41                  or what = protocol;
42     debug : group {
43        output : s=what / what = file or what = console or what = serial;
44        options : array of s;
45     }
46  # does anyone use this?
47     nvram : group {
48        media : s=what / what = file or what = sram or what = dram;
49     }
50  # these things are used by the engine
51     coordinates : group {
52        % : i=x,i=y,i=w,i=h / x+w <= 800, y+h <= 600;
53     }
54     infostrings : array of S;
55  # define a default item
56     % : free group ;   # a group type with no syntax group, but which can define
57                        # its own syntax group inside itself.
58  }
```

3. NEW .OTI FILE SYNTAX

The first part deals with the syntax of the `.oti` file itself. The second part covers how to retrieve data from the `.oti` file in C.

A. Very Abstract, Broad Overview

The `.oti` file provides a way to structure configuration data. The data is then easy to retrieve at run time using simple function calls. The file can also specify syntax, allowing dynamic type checking of data in the file.

B. Basic Structure Elements

Three structure elements are provided: "groups," "arrays," and "rows." "Rows" hold the actual data, consisting of integers, strings and real numbers (internal: real numbers even needed? Are other basic data types needed instead?). Arrays and groups just provide organization for the rows. An array is simply a homogeneous list of items, (rows, groups, or other arrays). Groups allow for a heterogenous collection of items, each one bound to a name.

C. Basic Typing

For each of these basic elements, a "subtype" of that element can be defined. After the resource manager loads a file, the groups, arrays, and rows, it then checks to see if each structure element matches its subtype.

The subtypes are defined either in file or in an auxiliary file. (internal: in fact, the specifics of this need to be worked out.) Rows can define how many data items they have and each of their types, i.e. string, integer, real. Rows can also specify formatting characters `<>()`, and restrictions on the values of the data in the row. Arrays types specify the subtype of their elements and restrictions on their size. Groups can give a list of names to be in the group and their subtype.

D. File Syntax

A row is a list of items separated by commas. The items are either numbers, strings or formatting characters. A row is ended with a semicolon. An example row:

```
5, seafood platter;
```

An array is enclosed in braces `{}`. The items in an array are not separated if they are rows, since the semicolon on the end of every row separates them well enough. Otherwise, the items are separated with commas, just for aesthetic reasons. An array of rows:

```
{ 5, seafood platter;
  458, light bulbs;
}
```

An array of arrays:

```
{
   { 1; 2; 3; },
```

```
    { 4; 5; 6; },
    { 123; 456; 789; }
}
```

Groups are also enclosed in braces {}. Groups are distinguished from arrays by the presence of bindings. A bindings is represented by a colon and it serves to unify a name with a value. A binding looks like:

```
<name> : <item>
```

The item is either a row, an array or another group. An example group:

```
{
   name : papa smurf;
   age : 150;
   parents : {
      mother : Betty;
      father : Barney;
   }
}
```

The file itself is treated as one group so that everything in a file is bound to a name.

E. Defining Subtypes

A row subtype is given as a list of format specifiers. The format specifiers are

```
i  <- integer
s  <- string
S  <- case sensitive string
f  <- real numbers
```

First a few notes. All strings specified as **s** are converted to lowercase. The **f** format character really stores its data as the C datatype **double** (and *not* as **float**).

These format specifiers are separated by commas. In addition, formating angle brackets and parenthesis can also be given, in pairs. An example row type:

```
S, (i,i), <i,i> ;
```

The type is ended with a semicolon. In addition, restrictions on the data values can be given. The first step is to give each data value of interest a name. This name is local only to this row type. Not every element needs a name. The name is assigned by following a format specifier with an equals sign and then the name of the variable. An example:

```
S, (i=x, i=y), <i,i>;
```

Now that we have variables we can use them in expressions. The available operators for expressions (from lowest to highest precedence):

```
Bool x Bool -> Bool
   ','
   and   or

Bool -> Bool
   not

NUMBER x NUMBER -> Bool
String x String -> Bool
   =   !=   <   <=   >   >=

NUMBER x NUMBER -> NUMBER
   +   -
   *   /

String -> Integer
   len()
```

The comma operator (under **Bool x Bool -> Bool**) is simply a very low-precedence **and** operator. This allows the writing of expressions such as

```
x>0, y>0
```

which allows one's eye to group the terms better. NUMBER is either Integer or Real. 'len' gives the length of a string. Parenthesis can be used to group sub-expressions. A restriction expression should evaluate to the boolean 'TRUE' if the data is okay. Any other value generates an error. Thus, the expression

```
1
```

would cause an error since it is the integer 1. The expression

```
1 = 1
```

evaluates to Boolean **TRUE**, and doesn't cause an error. The restriction expression is offset from the row definition by a forward slash. An example:

```
S, (i=x, i=y), <i=w, i=h> / x>0, y>0, w>0, h>0,
            x+w <= 800, y+h <= 600;
```

Two examples of valid rows based on this subtype:

```
What, (200,200), <100, 100>;
Are, (300, 500), <"100", 100>;
```

The second row is valid, even though the first 100 is in quotes because of how files are read in. Initially everything is read in as a string. When the subtype is applied to this row it tries to convert the strings which should be integers to integers. If it can't, an error is

raised. Now, three examples of an invalid rows based on this type:

```
You, (45, 67, 56);
Doing, 1,1,1,1;
"Here?", (500, 500), <200,200>;
```

The first row is invalid because it doesn't have enough elements. The second row doesn't have the right formatting. The third row has the correct format, but the data doesn't fit the restriction, since 500 + 200 > 600.

Array subtypes begin with the keyword **array of** followed by another subtype definition which gives the type of the elements in the array. An example:

```
array of i, s;
```

This gives an array of rows, each row consisting of an integer and a string. Any type definition is permitted, even

```
array of array of array of i,i;
```

Which defines an array of arrays of arrays of pairs of integers. To make it easier to see the grouping of arrays, braces can be used:

```
array of { array of { array of {i,i}}};
```

Notice that the semicolon moved to the outside of the braces. Arrays can also have restrictions. The restrictions are defined the same as they are for rows, except that arrays only have one variable: **length**. This is an integer giving the number of elements in the array. In addition, if an array has a restriction attached to it, the braces must be used. This prevents the restriction for the array from being confused with the restrictions for the row or other arrays. An example:

```
array of { array of i,i } / length > 5, length <= 10;
```

Group subtypes are identified with the keyword **group**. This is followed by an opening brace. Inside the brace is a list of bindings. These bindings associate a name with a type. An example:

```
group {
   city : s;
}
```

Associates the row subtype of a single string to the name **city**. The data would look like:

```
{
   city : New York;
}
```

The element bound to **city** in the data has the type bound to **city** in the definition. In addition, the name **city** is a mandatory binding and something must be bound to it in the data. An invalid group to the previous subtype is:

```
{
```

```
    town : New York;
}
```

Because there is no binding to **city**. Also invalid:

```
{
    city : New York;
    state : New York;
}
```

Because the subtype was not expecting a binding to **state**. This limitation motivates having optional bindings and default bindings. An optional binding may or may not appear in the data, a default binding is used for every name which the group subtype does not know how to deal with. An optional binding is indicated by putting a question mark before the binding name. Example:

```
group {
    ? city : s;
}
```

A default binding is indicated by binding to **%**. (internal: perhaps another keyword, e.g. "default"?). Example:

```
group {
    % : s;
}
```

Since the group type uses different name spaces for its mandatory bindings table and its optional bindings table it is possible to define

```
group {
    city : s;
    ? city : i;
}
```

In all cases, the group type first searches its mandatory binding table and then its optional binding table, and then finally it uses the default type, if there is a default type.

There is one more special kind of binding. A group type object allows you to define a type once and then use this type over and over. These are called "pure" types since no data can ever bind directly to them. Pure types are indicated by the keyword **type** before their name. Example:

```
group {
    type guess : i=low, i=high, i=guess / low <= guess, guess <= high;
}
```

No data objects can have this subtype of a group since it doesn't define any data bindings, it only has one pure binding. The definition can use this pure binding anywhere a subtype

is needed. Example:

```
group {
   type guess : i=low, i=high, i=guess / low <= guess, guess <= high;
   a : `guess;
   b : array of `guess;
   c : group {
      c : `guess;
   }
}
```

The back-quote makes a reference to the pure type. Then, when this reference is needed, the pure type is found and used. The references are not resolved at load time, only when they are needed. This means that the following passes without error since the optional binding to **dog** is not evaluated because there is no binding to **dog**:

```
subtype:
   group {
      ? dog : `dog_name;
      cat : s;
   }

data:
   {
      cat : whiskers;
   }
```

subtype definitions are identified by being bound to the name **syntax**. This means that the definitions must be the member of some group (any group may have a syntax binding, in fact). When a binding to syntax exists it is either deleted or it is used as the local syntax rules for parsing the group it is found in. The choice is up to the original subtype for the group with the syntax bindings. By default a group may not define its own syntax. To allow a group this freedom, the keyword **free** is placed before **group** in the subtype definition. The group definition for a free group is just the default subtype to use, in case the group does not have a syntax binding. An example:

```
subtype:
   free group {
      greeting : S;
   }

data:
   {
      syntax : group {
         % : i, i;
```

```
        }
        a : 1, 2;
        b : 3, 4;
    }
```

This is valid. The group subtype is declared **free**. The data has a binding to **syntax** giving alternate rules for this group. These rules are used to check the data.

An interesting point is that pure types are scoped and are available to all the subtypes in the group. The scoping should take into account the lazy evaluation of the references.

```
subtype:
    group {
        type city : group {
                name : S;
                where : `state;
        }
        type state : S;
        % : free group {
                dest : `city;
        }
    }

data:
    {
        first : {
            dest : {
                name : New York;
                where : New York;
            }
        }
        second : {
            syntax : group {
                type `state : S, i=zip;
                dest : `city;
            }
            dest : {
                name : Denver;
                where : Colorado, 80950;
            }
        }
    }
```

This is a contrived example, but it shows many things with type references. First, they are scoped. In the subtype definition of **city**, the reference to **state** is resolved in the parent. In the data, the group bound to **second** redefines the type **state** and uses a reference to **city**. The reference to **city** uses a reference to **state**, however the **state** type which is used is not the new redefined **state**, the type used for **state** is the one defined in the scope of the original definition of **city**. This behavior is similar to closures in Scheme.

Since subtype rules are identified by a binding to the name **syntax**, the element bound to **syntax** is a group subtype.

F. .oti File

The **.oti** file is treated as one large group. Thus, every element in the **.oti** file is bound to a name, and an **.oti** file may specify a syntax group to give new rules for it (but only if it is allowed).

Comments are started with a pound sign (#), and continue to the end of the line. To use the pound sign in a string, enclose the string in quotes.

G. Lexal Issues

All elements are read in from the file as a string. Only during the second step, rule checking, are the strings converted to integers or real numbers. This allows one to have strings of digits, and it also allows a number to be represented in many ways. Numbers can be written in decimal, hex (with a **0x** prefix), and octal (with a leading zero). Real numbers can be written in scientific notation (e.g. **-4.63e-28**). Strings can be written in quotes, either single or double. Quotes inside can be escaped with a back-slash. Examples:

```
"He said, 'Hello'"
'He said, "Hello"'
"He said, \"Hello\""
```

If you want something with the same name as a keyword, put it in quotes. Items in quotes are always identified as strings. In the following example the word "syntax" in quotes is not identified as the keyword **syntax**.

```
"syntax" : Something not giving rules;
```

H. Arcane Torture and File Grammar

The full grammar of the oti file is simple and boring. Here is the grammar of the restriction expressions, mildly more exciting. Notice that there is no provision for numbers. Strings are automatically coerced into integers and reals as needed. This allows both the string equality

```
s=string / string = "1";
```

and the integer equality

```
i=integer / integer = 1;
```

to be written as

```
s=string / string = 1;
i=integer / integer = "1";
```

although the second form is not very useful, and even a little misleading. The grammar of restrictions in the subtype definition:

```
restrictionlist := restrictionlist ',' restriction
        | restriction

restriction     := restriction AND r_term
                 | restriction OR  r_term
                 | r_term

r_term          := idop
                 | NOT idop

idop            := idterm '=' idterm
                 | idterm '!=' idterm
                 | idterm '<' idterm
                 | idterm '<=' idterm
                 | idterm '>' idterm
                 | idterm '>=' idterm

idterm          := idterm '+' idfactor
                 | idterm '-' idfactor
                 | idfactor

idfactor        := idfactor '*' idexp
                 | idfactor '/' idexp
                 | idexp

idexp           := STRING
                 | LEN '(' STRING ')'
                 | '(' restriction ')'
```

4. .OTI FILE ADDRESSES

The data in an **.oti** file is organized hierarchically. This allows us to define an address string to retrieve data. Elements in a group are identified by the name that they are bound to. Elements in an array are indexed by a 0 based non-negative integer in brackets. The addressing is very similar to the addressing of C structures and arrays. Consider the data:

```
boxes : {
```

```
      1, 1, 50, 50;
      100, 100, 50, 50;
   }
   arrayofgroups : {
      {
         name : Papa Smurf;
         age : 150;
      }, {
         name : Furby;
         age : 1;
      }
   }
   job : {
      type : busboy;
      wage : 8.00;
   }
   row : 1, 2, 3, 4, "time to go";
```

The last line has the address

```
row
```

The boxes are addressed as

```
boxes[0]
boxes[1]
```

In addition, arrays define an implicit element **length** which returns a single integer telling the number of elements in the array. This element would be addressed as

```
boxes.length
```

And we would get back 2. The groups are addressed as

```
arrayofgroups[0]
arrayofgroups[1]
```

The elements in the array of groups are addressed as

```
arrayofgroups[0].name
arrayofgroups[0].age
```

The items in job are addressed as

```
job.type
job.wage
```

Ultimately what is returned is a list of data — the list of data in the row.

APPENDIX E — MYGAME.STATE FILE

game.state file listing:

```
#
# GameState structure definition
#
# REQUIRED BY COMMON GAME ENGINE LAYER *******
# Common game configuration elements used by the engine -
# visible from the game layer... (Additional game specific
# elements should be created in a game specific configuration
# structure.)
#
struct configuration
  {
  char maxbet;
  char denomination;
  char bitmapped;
  char touchscreen;
  char credit_display;
  char bonus;
  char progressive;
  char network;
  int betinc;
  int red;
  int green;
  int blue;
  int cash_limit;
  int credit_limit;
  int handpay_limit;
  int cashout_limit;
  int volume;
  int payoutpercentage;
  int paytable;
  int hopper_fill;
  } Config;
# Local Game NVram
#
# All game specific components should reside here...
struct ball
{
int x;
int y;
int x_vel;
int y_vel;
int start_x_vel;
```

```
int start_y_vel;
int start_x;
int start_y;
char rectangle;
int count;
int current_rect;
} BBall;
struct bonus
  {
  int bonustrigger;
  } Bonus;
struct template
  {
  int working_example;
  } Template;
struct history
  {
  int example;
  } GameHistory[70];
struct winners
  {
  long rect_0;
  long rect_1;
  long rect_2;
  long rect_3;
  long rect_4;
  long rect_5;
  long rect_6;
  long rect_7;
  long rect_8;
  long rect_9;
  long rect_10;
long rect_11;
long rect_12;
long rect_13;
long rect_14;
  } Winner[4];
```

APPENDIX F — GENERIC GAME TEMPLATE FILE

1. GENERIC_TEMPLATE.C FILE LISTING

```
#include <userapi.h>

#include "game.h"

// include local game & graphics headers
#include "generic_template.h"



/******************/
/* Engine Hooks  */
/******************/

void init_game(void)
  {
  // Local data initialization
  }

void reset_game(void)
  {
    int volume;
  // Initialize all local game nvram
  nv_setchar("Config.denomination",(char)100/DENOMINATION);
  nv_setchar("Config.maxbet",(char)MAXBET);
  nv_setint("Config.betinc",(int)BETINC);
  nv_setchar("Config.bonus",(char)DOBONUS);
  nv_setint("Config.payoutpercentage",9201);

  nv_getint("Config.volume",&volume);
  sound_volume(volume);

  }

void begin_play(void)
  {
  // Local housekeeping and display cleanup
  nv_setint("Template.working_example",0);
  }

void play_one(void)
  {
```

```
  // First stage game animation and logic
  SetCurrentGameState((char)PLAY2);
  }

void play_two(void)
  {
  //Second stage game animation and logic
  SetCurrentGameState((char)EVALUATE);
  }

long evaluate_game(void)
  {
  // Compute payline award amount and return value
  return 0;
  }

void finish_game(void)
  {
  // Local housekeeping
  }

void bonus(void)
  {
  // Local pre-bonus logic & animation
  mod_load("bonus");
  }

void attract(void)
  {
  SetCurrentGameState((char)IDLE);
  }

char check_for_jackpot(void)
  {
  char jackpotlevel;
  jackpotlevel=0;
  return jackpotlevel;
  }

char check_for_bonus(void)
  {
  char bonusflag;
  bonusflag=0;
  return bonusflag;
  }

long bonus_complete(void)
  {
  return 0L;
  }
```

```
void animate_winner(void)
  {
  // Winner animations (ie - paylines...)
  }

void animate_idle(void)
  {
  // Idle mode animations
  }

void maxbet_game(void)
  {
  // Local maxbet logic and display routines...
  }

void cache_gfx(void)
  {
  // Load graphics into cache
  }

void pick_numbers(void)
  {
  int number;
  number=rnd_get_number(52);
  }

void draw_game_screen(void)
  {
  makebutton1("PLAY",510,530,60,60,GREEN,"spin_switch");
  makebutton1("BET" ,400,530,60,60,LTBLUE,"bet_switch");
  makebutton1("MAX",710,510,60,60,LTGREEN,"maxbet_switch");
  makebutton1("COLLECT",12,545,60,50,LTRED,"collect_switch");
  makebutton1("HELP",115,545,60,50,YELLOW,"help_switch");
  }

void update_lights(void)
  {
  }
```

2. GENERIC_TEMPLATE.H FILE LISTING

APPENDIX G — GRAPHICS CONVERSION TOOL

1. LOCATION AND USE OF CONVGFX.PY FILE

You will find the **convgfx.py** graphics conversion tool on the SGOS CD-ROM. ***Specify directory when known***

Refer to Chapter 14 for an overview of how to store and organize your graphic icons and use the **convgfx.py** tool to covert them to XPM format.

2. FILE LISTING

File Listing G-1 provides a listing of

File Listing G-1: convgfx.py

```
1   Place the final version of convgfx.py here
2
3
4
5
```

This file is included in the SGOS installation CD-ROM.

APPENDIX H — MAKESTRIPS UTILITY (9 LINE GAMES)

1. THE MAKESTRIPS UTILITY

Makestrips will turn a list of par-sheet files into C source code arrays. Since every par-sheet file has a different format, the task is complicated, but is still conceptually very simple:

- Read in the strip data from the par-sheet files.
- Format the data into a C header file.

Makestrips not only has many options governing how its default procedures do both steps, it will also let you replace either step with your own code.

To allow all this customization, Makestrips requires an auxiliary file to hold the settings of the various options. This file may have any name, but the default name Makestrips looks for is "makestrips.opt". If your file has another name you will need to specify it on the command line with the "-o <filename>" or "--optionfile <filename>" options. If you do not have an option file at all, you can either make one with a text editor, or run Makestrips with the "-n" or "--new" options. If you do the latter, you will still need to edit the resulting file to fill in important options, such as the par-sheet filenames.

Makestrips, by default, is lazy and checks the modification times of the option file and the input files against the modification time of the output file to see if the reel strips are already up-to-date. The check can be overridden by using the "-f" or "--force" options, or by setting the option [check mod times] in the option file to a blank line.

The header file generated by Makestrips has the format as shown in Figure [refrence to F:headfile]. The elements in angle brackets <> are user specified options. The options in the repeated block are represented in the option file as a list of values in the order they are to be used.

```
------------------------------------------------------------
/* [output file]
 * file automatically generated by makestrips.py
 * creation date and time
 */
#ifndef __[output file]
#define __[output file]
<head>
+-------------------------------------------+
|#ifdef [current percentage]
|#define PAYOUTPERC [current payout]
|[array name] = {
```

```
|   first reel strip,
|   second reel strip,
|   ...,
|   last reel strip
|};
|#endif
+-------------------------------------------+
          repeated for every [input file]
<tail>
#endif
-----------------------------------------------------------------
[figure F:headfile]
```

2. USING MAKESTRIPS

The command line syntax for Makestrips is:

```
./makestrips.py [-f | --force] [(-o | --optionfile) <filename>] [(-n | --new)
[<filename>]]
```

3. CUSTOMIZING MAKESTRIPS

Makestrips has many options which can be set, but options can only be set from an options file. Every option has a name which may include spaces and a corresponding value or list of values. Some options have defaults and do not need to be given specifically, although they can be assigned new values by specifying them. Other options do not have default values and must be included in the option file.

The list of options to be presented are organized according to what conceptual step they apply to: reading a par-sheet file, writing the C header file, or general options. An option description beginning with a '+' marks options which do not have default values and are required, otherwise the default values are given.

A. General Options

* [input files] '+' A list of par-sheet files. The order the files are listed is important, since the files are processed in the order listed. Their order should correspond with the order of the [percentages] and [payouts] options.

* [output file['+' The name of the C header file to be created (with extension).

* [check mod times] Set to either 0 or 1. 0 forces the reel strip file to always be recomputed. This is the same as the "-f" or "--force" command line options. If set to 1, Makestrips will only remake the reel strip file if the file modification times indicate that the reel strip file is out of date. The default value is 1.

B. Par-sheet Parsing Options

There are two reel strip parsers built into Makestrips: the default parser and the extended parser. Between the two of them just about any kind of paytable file can be read, but if these options are not enough, you can even write your own parser (see section [refrence to S:reelparser]).

The simple parser should be enough for most situations. It expects a tab delimited file and reads in the reel strips, one per column. The option [start column] adjusts the column the reel strips begin at. The extended parser, by contrast, uses regular expressions to interpret each line, and has the ability to go through a list of expressions until it finds one that matches.

Common Parser Options. The two parsers share a few options. These options have the same meaning with both parsers.

* [start line] A regular expression describing a line before the reel strips. The lines of each input file are read and discarded until a line matching [start line] is found. Reel strip parsing begins with the next line. If nothing is given, the parsing begins with the first line of the file.

* [stop line] A regular expression describing the line to stop on. Once parsing has begun, every line is checked against [stop line]. When a line matches the line is discarded and reel strip parsing of the file stops. If nothing is given, the parsing goes through every line of the file.

* [num reels] The total number of reels. The default value is 5.

* [extended parser] If set to 1, the extended parser will be used. The default value is 0---use the simple parser.

Simple Parser. The simple parser assigns each reel to a column of the input file. It is important to specify [start line] and [stop line] so that the parser won't put whatever extraneous garbage that is also in that column into the reel strip.

* [start column] An integer giving the column the first reel strip is in. The other strips are assumed to be in the next [num reels] columns. This option is zero based, making the first column `column 0'. The default value is 0.

Extended Parser.The extended parser uses regular expressions. It starts with the pattern given in [line pattern 1]. If the current line matches the pattern, the reel values are pulled from the line by grouping marks in the pattern. The groups are mapped into the reel strip arrays by the option [line map], with the first group going to the first reel strip index in [line map], the second group to the second index in [line map], etc. The pattern matching continues until a line doesn't match the current line pattern. If the option [line pattern 2] is given, [line pattern 2] become the new pattern and the current line is re-parsed with it, otherwise the current line is skipped, the pattern is not changed and the next line is read. Repeat this process for each line pattern in the option file. Whenever a new line pattern is loaded, the next [line map <n>] is searched for. If no line mapping is given, the previous one is used.

* [line pattern <n>] '+' A list of regular expression patterns. There should be grouping elements `()' in the expression, one for each value to be extracted. The extracted val-

ues are put in left to right order into the strips for each reel. If there are more groups than reels, an error occurs. If [line pattern] is a list of expressions, the expressions are joined together with the regular expression =+=, which will match anything that is not alphanumeric or `_' such as spaces, tabs and commas. Only required if [extended parser] is set to ``1".

* [reel map <n>] '+' A list mapping the grouping elements in [line pattern <n>] to reel strips. Only required if [extended parser] is set to ``1".

C. C Header File Options

* [head] A list of text to insert at the beginning of the header file. Since Makestrips already inserts its own header, [head] would follow.

* [tail] A list of text to be included at the end of the header file. [tail] appears before Makestrips's tail.

* [percentages] The list of percentages. "payouts" - the list of percentages and payouts.

4. THE FORMAT OF THE OPTIONS FILE

The most important item in an option file is a tag, which names a specific option. A tag begins with a left bracket, `[', in the first column, and ends with the first right bracket `]' on the line. All the text between the brackets is the option tag; all text remaining on the line is ignored. After a tag line, the lines after the tag are read in, even blank lines, and stored associated with the option given by the tag. An exception is the blank lines between the end of a tag's items and the next tag: these blank lines are ignored. However, the blank lines occurring between a tag's items are stored. The text before the first tag in the file is also ignored, allowing a nice little area for comments. The only other space for comments in the file are on a tag line after the tag.

A special exception is given to the text following the the tags [start line], [line pattern], and [stop line]. These tags require regular expressions, and need to use the left bracket, `[', character. A single space placed at the start of a line will be striped out, allowing regular expressions to begin with a left bracket `['. If your regular expression needs to begin with a blank space, start the line with two blank spaces since only the first one is removed.

The order the options are given in the file are unimportant.

5. WRITING A PAR-SHEET PARSER

In case the two parsers built into Makestrips are not adaptable to read your par-sheet file, it is possible to write your own file parser. The parser should be a code snippet (not a function definition), and be included in the option file under the option [user parser]. The code snippet is expected to read in the option file and make assignments to the variable "reels", which is a list containing as many lists as reels. (As given by the option [num reels].) The first reel has index 0, the second reel index 1, etc. This variable will then be written to [output file] by Makestrips. Your code snippet can use the following implicit variables: "reels", "filein", "stop_pat", and "get_option()", as well as using any builtin variables and importing as many modules as it likes. No restrictions are placed on the execution environment. The variable "reels" is the variable you should assign strings to, this is the variable printed into the header file. "filein" is an

already open file object. The file position pointer has already been advanced to the first line after the line described by [start line]. "stop_pat" is a regular expression describing the line to end on. Your code does not need to make use of it. "get_option("<option name>")" is a function which will return the value of the option <option name>, which is passed as a string. The option will be returned as a list of strings, with each string being a separate line in the option file. Although your parser will be invoked once for every par sheet file, there are no provisions for the script to pass variables between invocations.

As an example, here is the simple default parser as it would look in the option file if it were written as a code snippet.

```
[user parser]
import re
import string
start_col = int(get_option('start column')[0])
num_reels = int(get_option('num reels')[0])
end_col = start_col + num_reels
for l in filein.readlines():
    if stop_pat and re.match(stop_pat,l):
        break
    tok = string.split(l[:-1],'')
    i = 0
    for sym in tok[start_col:end_col]:
    if sym != '':
        reels[i].append(sym)
    i = i + 1
```

APPENDIX I — NINE LINE GAME TEMPLATE

1. NINELINE_TEMPLATE.C FILE LISTING

add when available

2. NINELINE_TEMPLATE.H FILE LISTING

add when available

APPENDIX J — POKER GAME TEMPLATE

1. POKER_TEMPLATE.C FILE LISTING

add file lising when available -- this is font

APPENDIX K — OTHER TEMPLATES

these templates are under construction

1. HELP TEMPLATE
2. LAST GAME TEMPLATE
3. PAY TABLE TEMPLATE
4. BONUS TEMPLATE

APPENDIX L — ONLINE PROTOCOL EXCEPTION CODES

Exceptions.h is a complete list of olga (Shuffle Master's Online Gaming Architecture) exception codes. With the relevant exception codes below olga will handle exceptions for sas, sds, and ogp protocols.

exceptions.h listing:

```
/*
   Shuffle Master Game Library
   Copyright (c) 1999-2000 Shuffle Master, Inc.
   All Rights Reserved.

   $Revision: 1.5 $
   $Author: mdj $
   $Date: 2000/07/31 22:00:23 $
*/

#ifndef __EXCEPTIONS_H
#define __EXCEPTIONS_H

// these defines are for all possible exceptions
// door events
#define MAINDOOROPENED0x01
#define MAINDOORCLOSED0x02
#define DROPDOOROPENED0x03
#define DROPDOORCLOSED0x04
#define LOGICDOOROPENED0x05
#define LOGICDOORCLOSED0x06
#define CASHDOOROPENED0x07
#define CASHDOORCLOSED0x08
#define BELLYDOOROPENED0x09
#define BELLYDOORCLOSED0x0A
#define NOTEDOOROPENED0x0B
#define NOTEDOORCLOSED0x0C
#define DOOR7OPENED0x0D
#define DOOR7CLOSED0x0E
#define DOOR8OPENED0x0F
#define DOOR8CLOSED0x10

// hopper & coin events
#define HOPPERFULL0x12
#define HOPPERLOW0x13
#define HOPPEREMPTY0x14
#define OVERRUN0x15
#define COINOUTERROR0x16
#define COININERROR0x17
#define DIVERTERERROR0x18
#define REVERSECOIN0x19
#define COINLOCKOUTFAIL0x1A
```

```
#define COINOUTJAMOx1B
#define COINDROPOx1C
#define TOKENDROPOx1D
#define CASHOUTCOINSOx1E
#define CASHOUTTOKENSOx1F
#define WINCOINSOx20
#define WINTOKENSOx21
#define WINCREDITSOx22
#define CREDITSFROMCOINOx23
#define CREDITSFROMTOKENOx24

// reel events
#define REELTILTOx27
#define REEL1TILTOx28
#define REEL2TILTOx29
#define REEL3TILTOx2A
#define REEL4TILTOx2B
#define REEL5TILTOx2C
#define REEL6TILTOx2D
#define REELDISCONNECTOx2E
#define REELSTOPPEDOx2F

// bill and stacker events
#define ACCEPTORFAILROMOx32
#define ACCEPTORFAILCSOx33
#define ACCEPTORFAILOx34
#define ACCEPTORJAMOx35
#define STACKERJAMOx36
#define REVERSEBILLOx37
#define BILLREJECTEDOx38
#define BILLCOUNTERFEITOx39
#define STACKERFULLOx3A
#define CASHBOXREMOVEDOx3B
#define CASHBOXINSTALLEDOx3C
#define BILLIN1Ox3D
#define BILLIN2Ox3E
#define BILLIN5Ox3F
#define BILLIN100x40
#define BILLIN200x41
#define BILLIN500x42
#define BILLIN1000x43
#define BILLIN2000x44
#define BILLIN5000x45
#define BILLIN1CHANGEOx46
#define BILLIN2CHANGEOx47
#define BILLIN5CHANGEOx48
#define BILLIN10CHANGEOx49
#define BILLIN20CHANGEOx4A
#define BILLIN50CHANGEOx4B
#define BILLIN100CHANGEOx4C
#define BILLIN200CHANGEOx4D
#define BILLIN500CHANGEOx4E
#define BILLIN$480x4F
```

```
// printer events
#define PRINTEROFF0x52
#define PRINTERON0x53
#define NEEDRIBBON0x54
#define CARRIAGEJAM0x55
#define PRINTERCOMM0x56
#define PAPERLOW0x57
#define PAPEROUT0x58

// user generated events
#define WAITINGFORUSER0x5B
#define CANCELHANDPAY0x5C
#define CASHOUTBUTTON0x5D
#define BUFFERFULL0x5E
#define CHANGEREQUESTED0x5F
#define GAMESTOP0x60
#define DRAWCARDS0x61
#define BET0x62
#define CARDHELD0x63
#define GAMESELECTED0x64
#define GAMESTART0x65
#define GAMESTARTCOIN0x66
#define GAMESTARTCREDIT0x67
#define GAMESTARTTOKEN0x68

// attendant generated events
#define CHANGECANCELED0x6B
#define BILLTOTALSRESET0x6C
#define OPTIONCHANGE0x6D
#define JACKPOTRESET0x6E
#define DISPLAYMETERS0x6F
#define DISPLAYMETERSEXIT0x70
#define SELFTESTSTART0x71
#define SELFTESTEND0x72
#define RECALL0x73
#define SOFTMETERSRESET0x74

// memory events
#define RAMRECOVERED0x77
#define RAMCLEARED0x78
#define RAMBAD0x79
#define ROMERROR0x7A
#define ROMBAD0x7B
#define ROMCHECKSUMCHANGE0x7C
#define ROMCHECKSUMERROR0x7D
#define PROMCHECKSUMCHANGE0x7E
#define PROMCHECKSUMERROR0x7F
#define MEMORYRESET0x80
#define BATTERYLOW0x81

// progressive
#define PROGRESSIVELINKFAIL0x84
#define PROGRESSIVEJACKPOT10x85
```

```
#define PROGRESSIVEJACKPOT2Ox86
#define PROGRESSIVEJACKPOT3Ox87
#define PROGRESSIVEJACKPOT4Ox88
#define PROGRESSIVEJACKPOT5Ox89
#define PROGRESSIVEJACKPOT6Ox8A
#define PROGRESSIVEJACKPOT7Ox8B
#define PROGRESSIVEJACKPOT8Ox8C
#define SASPROGRESSIVEOx8D

// other events
#define POWERUPOx90
#define POWERDOWNOx91
#define GENERALTILTOx92
#define CASHOUTTICKETOx93
#define HANDPAYJACKPOTOx94
#define HANDPAYCREDITSOx95
#define CASHOUTOx96
#define RESETDURINGPAYOUTOx97
#define BONUSPAYOx98
#define OUTOFSERVICEOx99
#define TOUCHERROROx9A
#define SIGNERROROx9B
#define PROCESSORRESETOx9C

#endif
```

APPENDIX M – SCREENS FOR SETUP AND RECORDKEEPING

1. MAIN SCREEN

SGOS provides default screens for several setup, recordkeeping and diagnostics features. The following screen offers eight further options:

- Machine Statistics
- Ticket-Bill History
- Last Games
- Location Information
- Exit
- Test
- Configuration Guide
- Out of Service

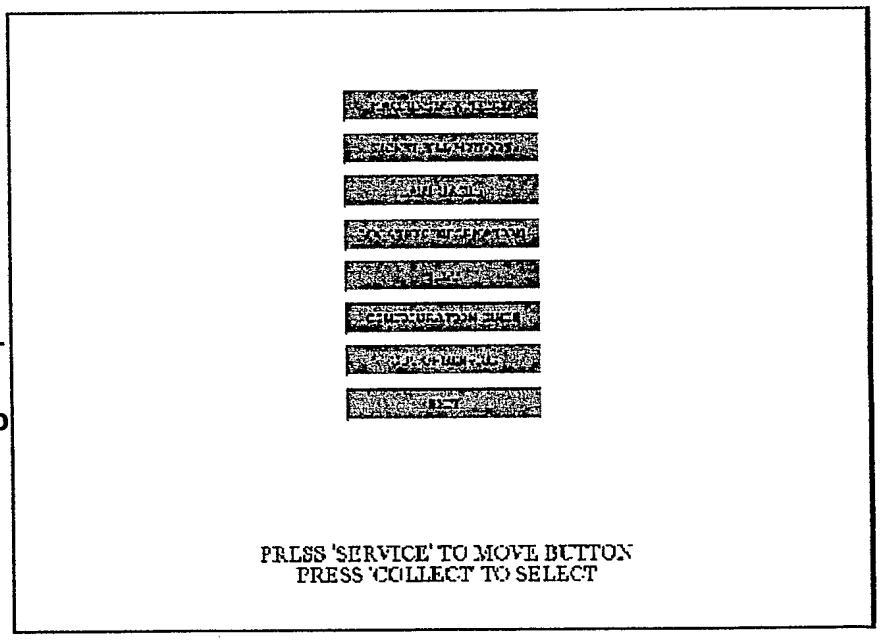


Figure M-1 – Setup, Recordkeeping and Diagnostics Main Screen

2. MACHINE STATISTICS

The "Machine Statistics" button on the main screen leads to the two screens shown in Figure 7 and Figure 8. The items tracked in Figure 7 are as follows:

"Soft Meter" ItemExplanation
Coins InNumber of credits from coins

Bills In Number of credits from bills
Drop Number of credits in the drop (coins only)
Total In Simple count of total credits in
Total OutSimple count of total credits out
Credits PlayedTotal of all credits played
Credits Won Total credits won by playing, excluding jackpots and hand pay
Credits Paid Credits paid from hopper, excluding hand pays or jackpots
Games PlayedSimple count of games played
Games WonSimple count of games won
Hand PayTotal hand pays in credits, including errors paid by attendant
Jackpot Total jackpots in credits, all attendant paid
Hit frequency% of played games that win
Awarded %Payback percentage

3. ERROR COUNTS

The error counts in the following screen are all simple counters as noted. (Each door opening counts as an error.) Games played are also simple counts since the last power up or door closed events. Counters max out at 1000 and remain there until the next reset/clearing event occurs.

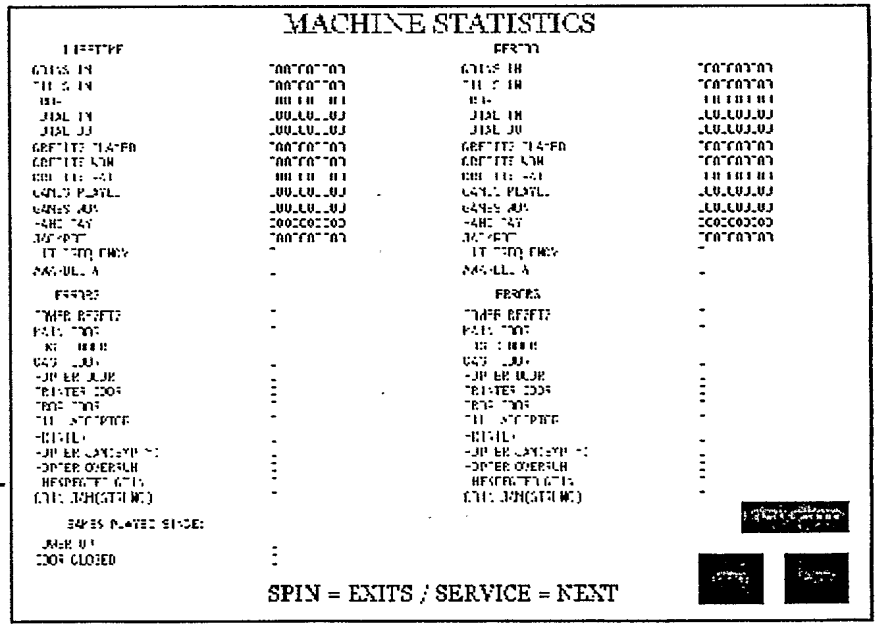


Figure M-2– Machine Statistics Screen #1

Pressing the "Next" button above leads to the following screen which shows the number of wins for each possible payline.

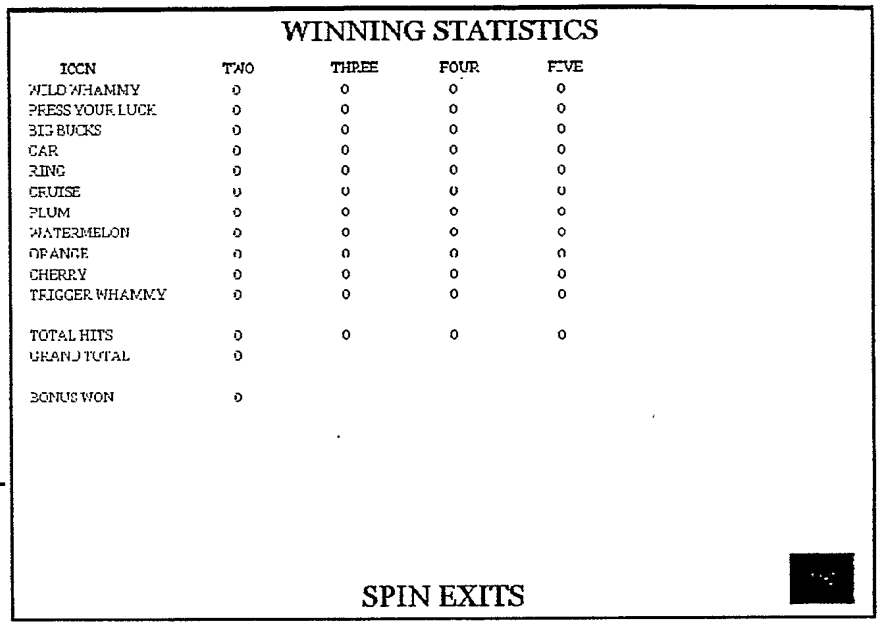


Figure M-3– Machine Statistics Screen #2

4. TICKET-BILL HISTORY

The "Ticket-Bill" button on the main screen leads to the following two screens. The Bill History Screen gives the date and time of the last twenty bills accepted, for machines with bill acceptors, and a history of tickets printed where applicable.

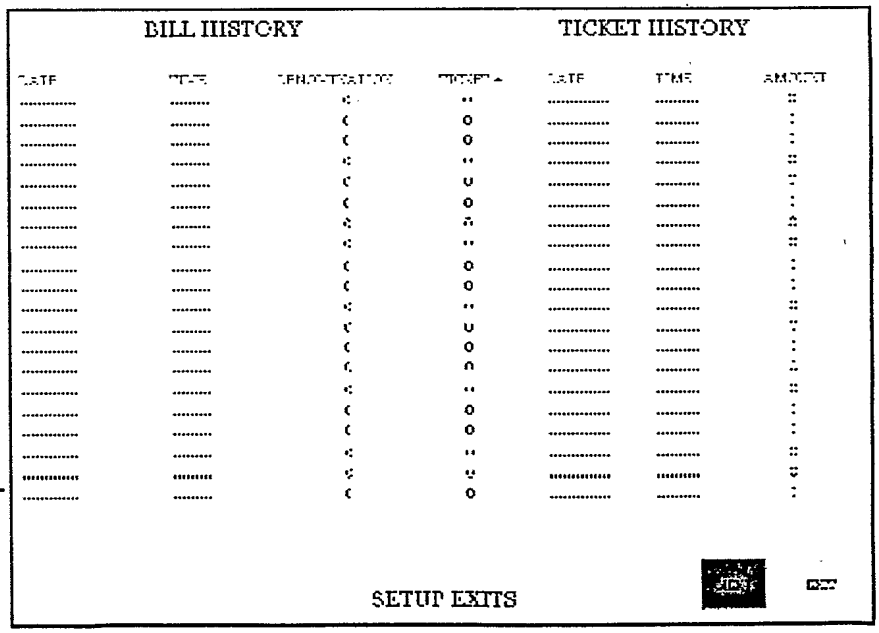


Figure M-4– Ticket Bill History Screen #1

The Stacker/Hopper Inventory Screen gives a count of bills and coins in the machine (as applicable). The screen items are as follows:

Screen Item	Explanation
Stacker Hopper Inventory	Accumulates amounts in increments shown
Stacker Value	Gives total value of bills, amount as credits, and actual number of bills instacker
Hopper Value	Shows credits in hopper (actual coins); fill amount must match bag amount

ButtonsAttendant can adjust the fill amount with the "increase/decrease" buttons.

Standard fill amount is set in the Configuration Screen (Figure 18).

Attendant selects current fill amount with the "add/remove" buttons.

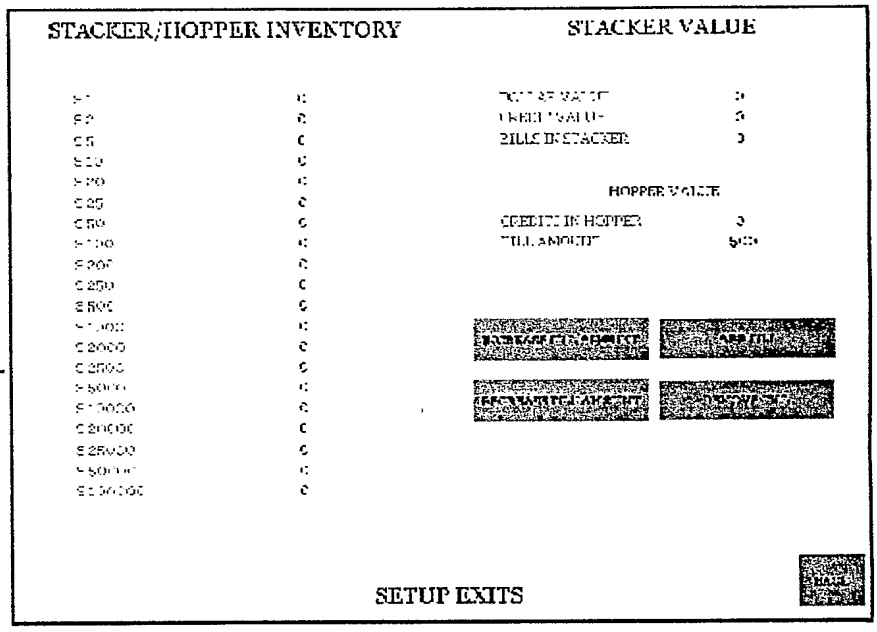


Figure M-5– Stacker/ Hopper Inventory (Ticket Bill History Screen #2)

The Game History Screen in Figure 11 provides a 70-step game history for resolving player disputes. The example below is from a 9-line game. (How/where code in graphics for developer's game?)

5. LOCATION INFORMATION

The Location Information screen accepts basic user data.

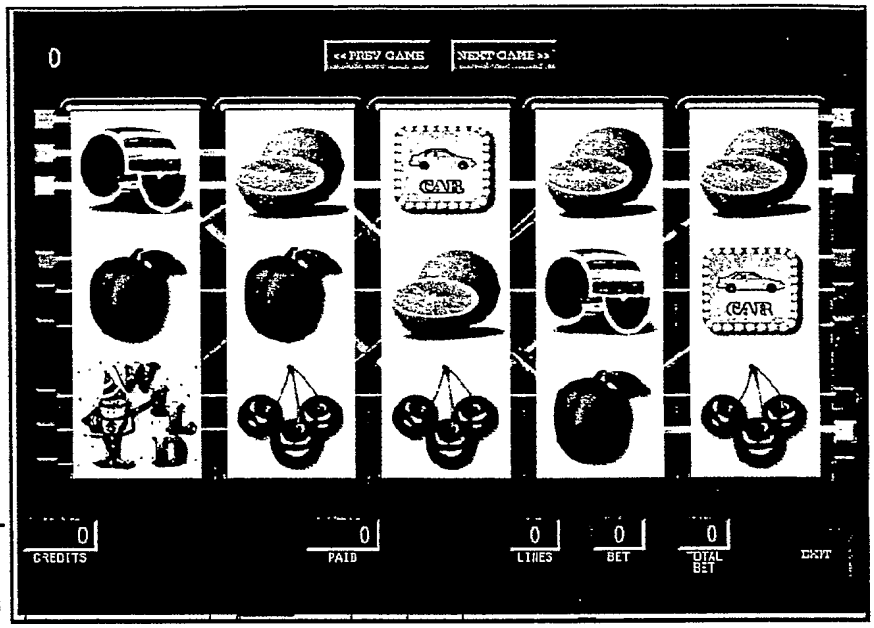


Figure M-6 – Game History Screen

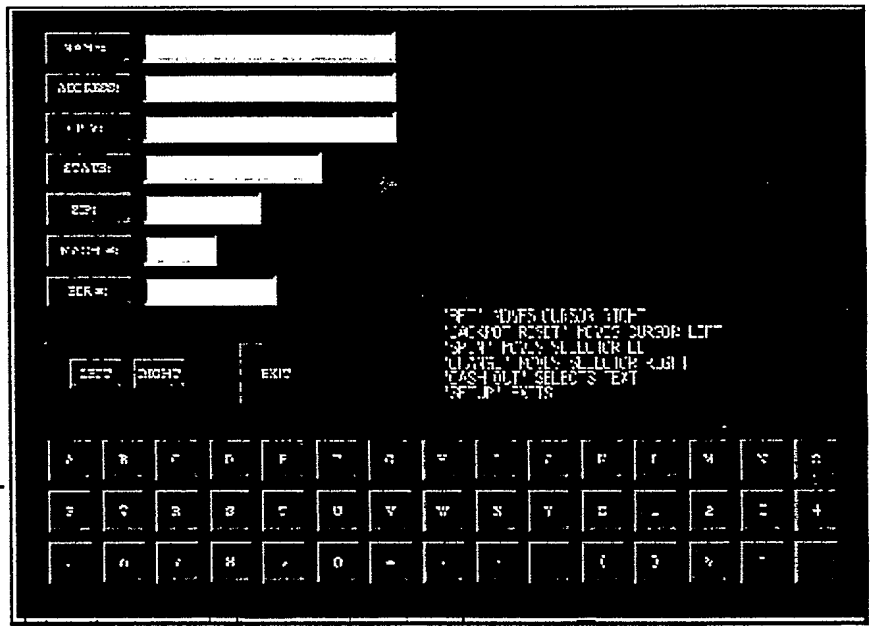


Figure M-7 – Location Information Screen

6. Diagnostic Test Screens

The Diagnostic Tests screen runs five tests as shown in the following screens:

1. Lights/Switches
2. Touchscreen
3. Bill Validator
4. Hopper
5. Sound (set volume)

At the start of the Light/Switch Test all switches show "unknown." To test, the attendant or

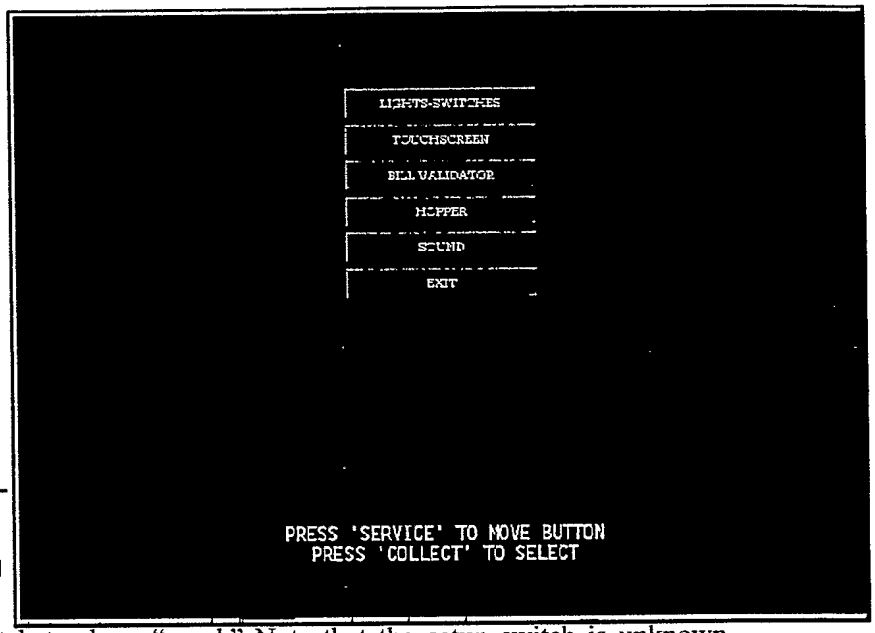


Figure M-8 – Diagnostic Tests Menu Screen

slot tech must press each switch to show "good." Note that the setup switch is unknown because it is used to exit this menu.

The touch screen test displays instructions on the screen.

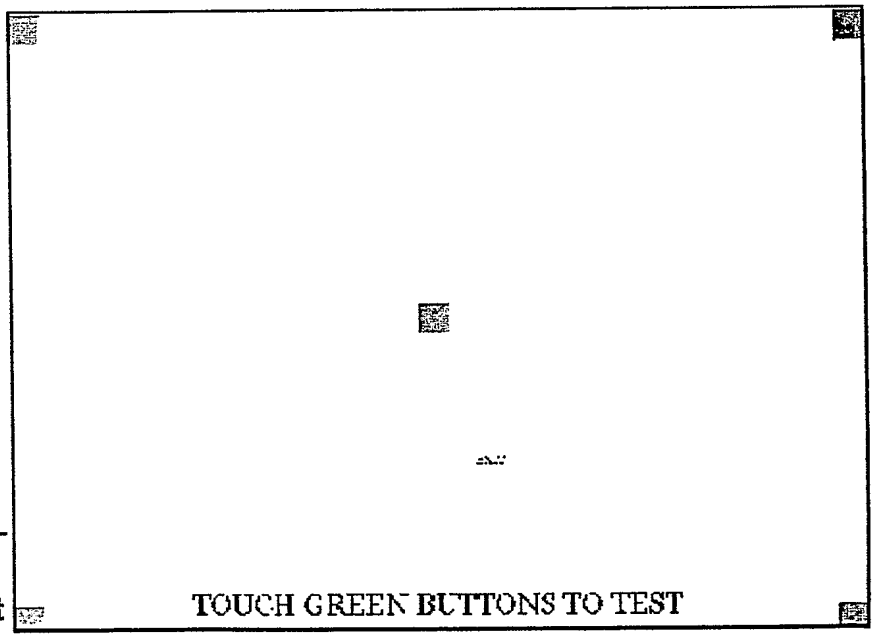
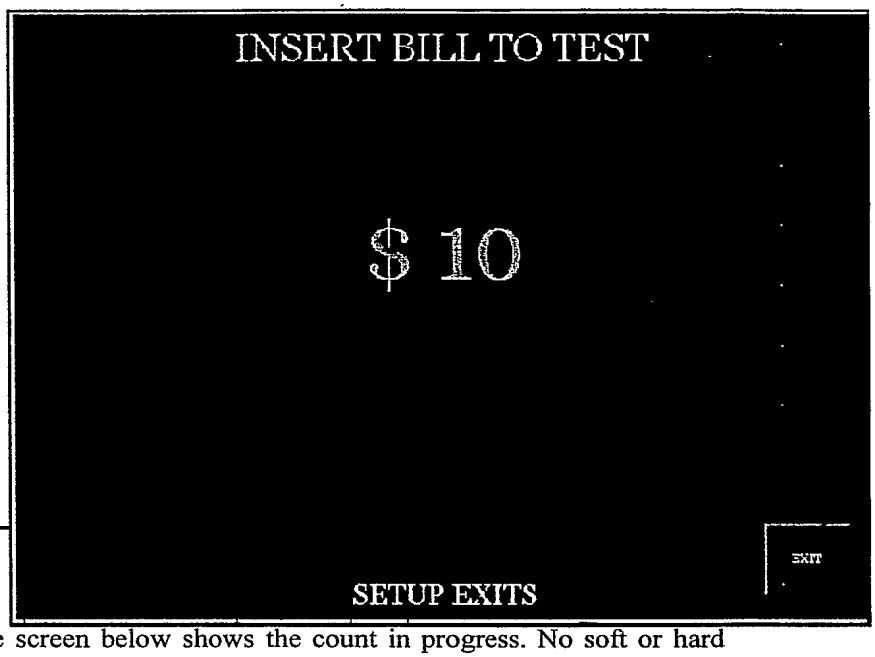


Figure M-9 – Touch-screen Test

The bill test screen verifies the bill acceptor operation. The bill is returned after the bill

denomination is identified. No soft or hard meters are affected by the insertion of bills.

The Hopper Test confirms with pressing the "Start" button that the hopper dispenses coins cor-



FigureM-10– Bill Test Screen

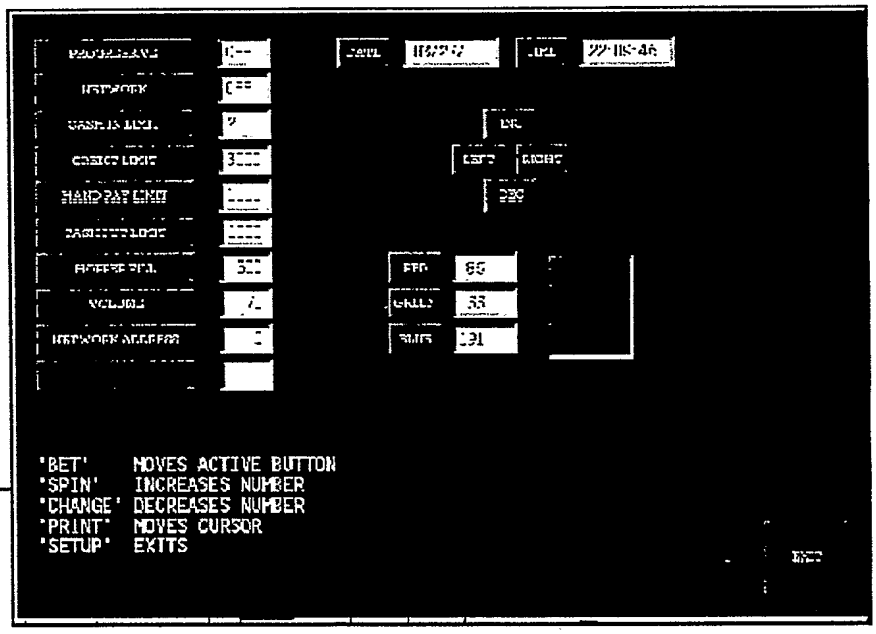
rectly. Test is for 20 coins; the screen below shows the count in progress. No soft or hard meters are affected by the dispensing of coins.

FigureM-11– Hopper Test Screen

7. CONFIGURATION GUIDE

The Configuration screen sets the following:

Configuration Setting	Explanation
Progressive	Whether jackpots are progressive
Network	Should be "on" when applicable
Cash in Limit	Sets max credit player can accumulate before they must play
Credit Limit	Automatically pays any amount above setting
Hand Pay	Any credits won beyond this setting must be paid during cashout
Hopper Fill	Standard hopper bag amount
Volume	Speaker volume
Network Address	Ref number for this machine (for network, where relevant)



FigureM-12–Game Configuration Screen

APPENDIX N – ADVANTEC HARDWARE SOLUTION INFORMATION

1. CHIMP – PCM-5864 EMBEDDED CONTROLLER

A. Flow/Block Diagram

Not available from Shuffle Master, please contact Advantec.

B. Schematic Diagram

Attached

C. Layout Diagram

Please refer to page 7 (Acrobat page 18) of the PCM-5864/L Users Manual, available from Advantec. (PCM5864-L.pdf)

D. Parts List

Not available from Shuffle Master, please contact Advantec.

E. Jumper Settings

Please refer to pages 9-37 (Acrobat pages 20-48) of the PCM-5864/L Users Manual, available from Advantec. (PCM5864-L.pdf)

Jumper	Name	Settings	Selection
JP1	ATX power switch	All pins open	Not used
JP2	Watchdog Action	Short 1-2, Open 3	System Reset
J3	CPU voltage	Open 1-2, Short 3-4, Open 5-6, Open 7-8	2.20V
J5	CPU frequency ratio	Short 1-2, Short 3-4, Short 5-6	4.5
J6	Reserved	All pins open	
J7	CMOS Clear	Short 1-2, Open 3	Battery On
J8	LCD Backlight	Short 1-2, Open 3	Positive
J9	System/PCI clock	Short 1-2, Open 3-4, Short 5-6	66MHz/33.3MHz
J10	PCI/Clock	Open 1, Short 2-3	33MHz
J11	COM4 RI	Open 1-2, Open 3-4, Short 5-6	RI
J12	COM3 RI	Open 1-2, Open 3-4, Short 5-6	RI
J13	Compact Flash	Installed	Enabled (don't care)
J14	COM1 RI	Open 1-2, Open 3-4, Short 5-6	RI

J15	COM2 RI	Open 1-2, Open 3-4, Short 5-6	RI
J16	LCD Power	Short 1-2, Open 3	+5V
J17	Reserved	All pins open	
J18	Buzzer	Installed	Enabled
J19	COM2 Mode	Short 1-2, Open 3-4, Open 5-6	RS-232
J20	COM2 Mode	Short 1-3, Short 2-4, Open 5-6	RS-232
J21	COM2 Mode	Short 1-3, Short 2-4, Open 5-6	RS-232
J22	LCD Shift Clock	Short 1-2, Open 3	SHFCLK
J23	Audio Power	Open 1, Short 2-3	+12V
J25	TV-Out	Open 1, Short 2-3	NTSC

F. Removable Components

BT1 – Battery for CMOS configuration settings.
U18 – BIOS ROM
U529 – CPU
DIM1 – Dynamic RAM memory module.

G. Connections

Label	Function	Status
CN1	CPU Fan	Not Installed
CN2	Motherboard Fan	Not Installed
CN3	Ethernet	Not used – May be cabled to HABIT
CN4	Audio	Cabled to HABIT
CN5	PCI	Not used
CN6	CD Audio IN	Not used
CN7	AUX Line In	Not used
CN8	Main Power	Cabled to HABIT
CN9	Keyboard & Mouse	Not used – May be cabled to HABIT
CN10	Floppy Drive	Not used
CN11	PC/104 ISA bus Expansion	CARD STACK
CN12	IDE Hard Drive	Not used
CN13	Reserved	Not used
CN14	Parallel Port	Cabled to HABIT
CN15	Front Panel	Not used

CN16	USB	Not used – May be cabled to HABIT
CN17	IR	Not used
CN18	CRT Display	Cabled to HABIT
CN19	Video Out	Not used
CN20	Flat Panel	Not used
CN21	Ext. Flat Panel	Not used
CN22	Peripheral Power	Not used
CN23	COM Ports	Cabled to HABIT
CN30	Video In	Not used
CN501	Compact Flash	Not used
CN502	Speaker Out	Not used
DIM1	DIMM	Dynamic Ram
J1	ATX Feature	Not used
JP3	LVDS	Not used

2. PCM-3810 RAM CONFIGURATION

A. Flow/Block Diagram

Not available from Shuffle Master, please contact Advantec.

B. Schematic Diagram

Attached

C. Layout Diagram

Please refer to page 1 of the PCM-3810A PC/104 Solid-State Disk Module manual, available from Advantec.

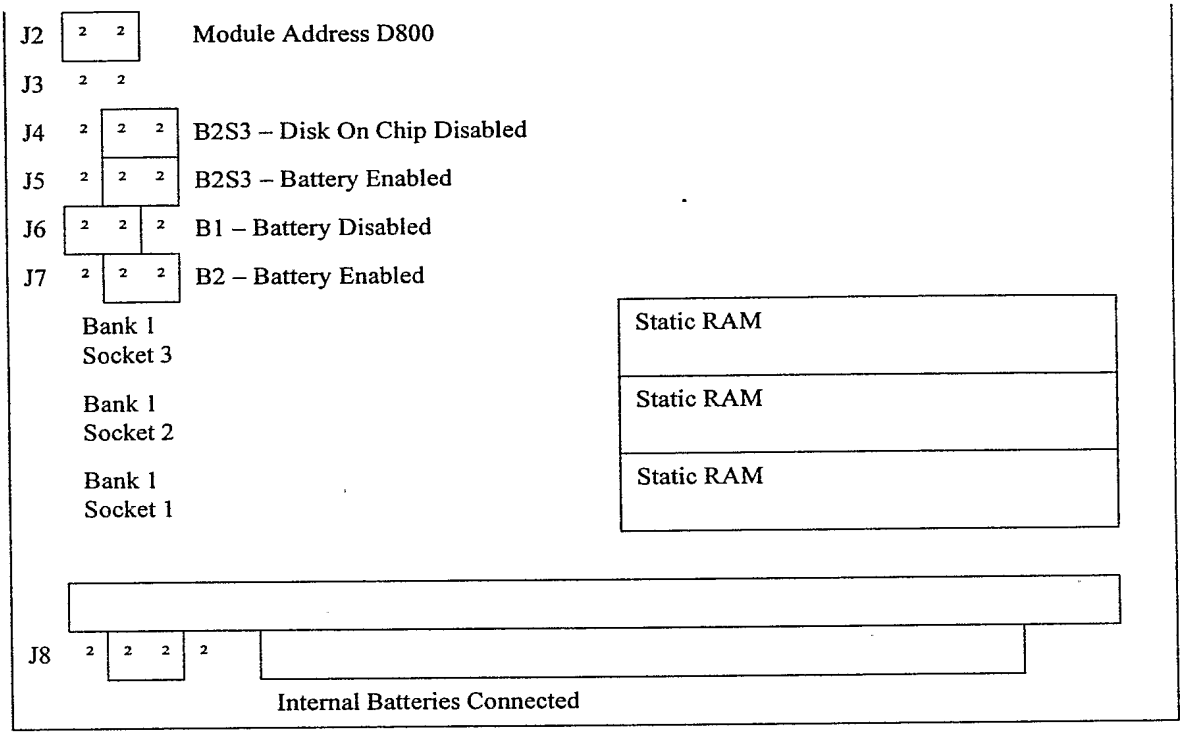
D. Parts List

Not available from Shuffle Master, please contact Advantec.

E. Jumper Settings

Strapping for the "PCM-3810A PC/104 Solid State Disk Module." Battery backed Static RAM configuration.

J1 2 2



F. Removable Components

M1 - Bank 1, Socket 1 – not currently used.
M2 - Bank 1, Socket 2 – not currently used.
M3 - Bank 1, Socket 3 – not currently used.
M4 - Bank 2, Socket 1 – Static RAM
M5 - Bank 2, Socket 2 – Static RAM
M6 - Bank 2, Socket 3 – Static RAM
BT1 - Battery - Installed
BT2 - Battery - Installed

G. Connections

PC/104 Connector – PC bus
CN1 – Reserved by Advantec.

3. PCM-3810 OS/DOC Configuration

A. Flow/Block Diagram

Not available from Shuffle Master, please contact Advantec.

B. Schematic Diagram

Not available from Shuffle Master, please contact Advantec.

C. Layout Diagram

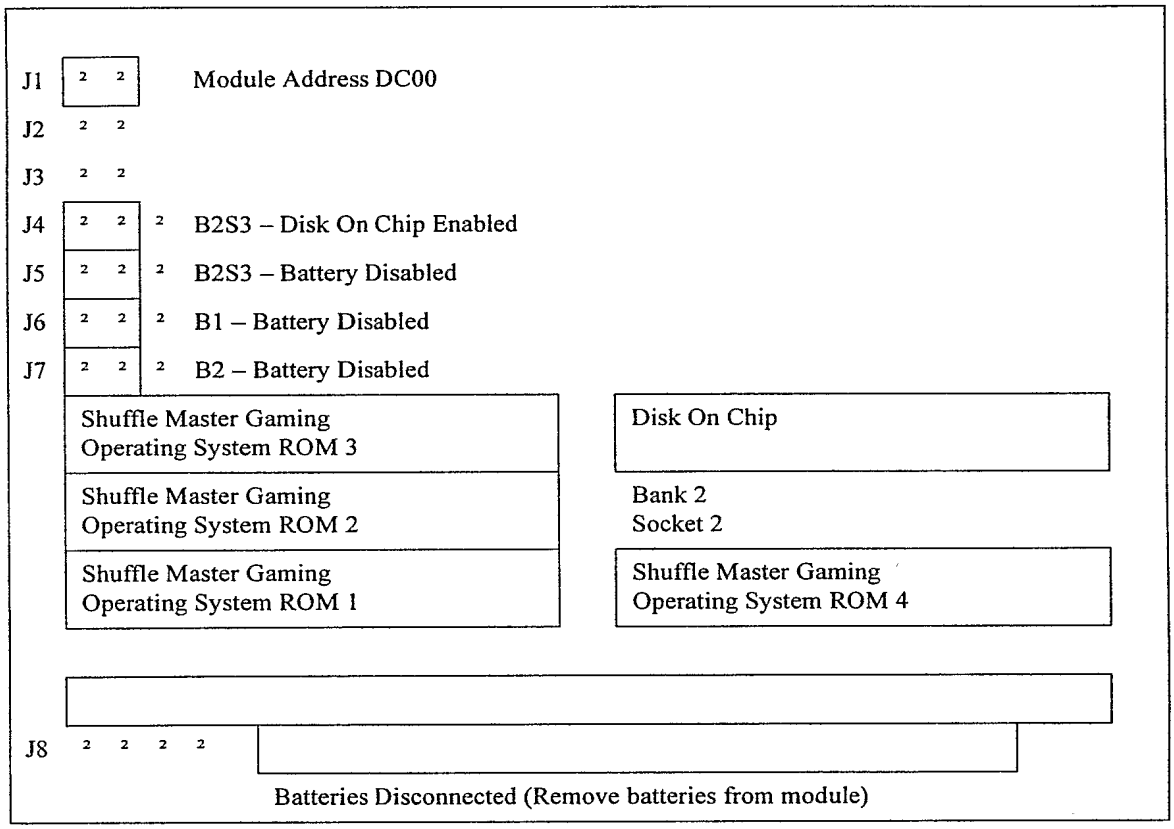
Please refer to page 1 of the PCM-3810A PC/104 Solid-State Disk Module manual, available from Advantec.

D. Parts List

Not available from Shuffle Master, please contact Advantec.

E. Jumper Settings

Strapping for the "PCM-3810A PC/104 Solid State Disk Module." Operating System and Disk on Chip configuration.



F. Removable Components

Bank 1, Socket 1 – SGOS ROM # 1
Bank 1, Socket 2 – SGOS ROM # 2
Bank 1, Socket 3 – SGOS ROM # 3
Bank 2, Socket 1 – SGOS ROM # 4
Bank 2, Socket 2 – not currently used.
Bank 2, Socket 3 – Disk On Chip – Game personality program.
Batteries – Removed

G. Connections

PC/104 Connector – PC bus
CN1 – Reserved by Advantec.

4. HIC

A. Flow/Block Diagram

Attached "HICflow.vsd"

B. Schematic Diagram

Attached

C. Layout Diagram

Attached

D. Parts List

Attached

E. Jumper Settings

One jumper that selects SRAM secondary enable from either Vcc or Battery Monitor. Currently the jumper is set to select Vcc by shorting pins 1-2, pin 3 is open. (Jumper away from the connector.)

F. Removable Components

U2 – Control Decode GAL
U3 – Address Decode GAL
U10 – Watchdog ROM

G. Connections

JP1, JP2 – PC/104 Connector – PC bus
P1 – I/O to HABIT

5. HABIT

A. Flow/Block Diagram

Attached "HABITflow.vsd"

B. Schematic Diagram

Attached

C. Layout Diagram

Attached

D. Parts List

Attached

E. Jumper Settings

No jumpers, however there is an ECO to remove the non-working power off door monitor circuit.

See Engineering Change Orders FTC-001, FTC-002, and FTC-003, available from Advantec.

F. Removable Components

There are no removable components.

G. Connections

P1 – Game Harness
P2 – Game Harness
J1 – Feed through COMM ports from CHIMP
J4 – from Power Supply
J8 – Audio from CHIMP
J9 – Feed through Printer from CHIMP
J10 – Feed through USB from CHIMP
J11 – Feed through Keyboard & Mouse from CHIMP
J12 – Feed through Ethernet from Chimp
J13 – Feed through Video from CHIMP
J14 – Power for CHIMP
J16 – I/O from HIC

APPENDIX O – FURTHER HELP AND TROUBLESHOOTING

1. SHUFFLE MASTER WEBSITE

shufflemastersupport.com

2. TROUBLESHOOTING

[add list of known problems and solutions]

What is Claimed is:

1. A method of assisting in the development of a computer based wagering gaming application comprising:

providing a gaming operating system comprising a library of at least two software gaming callback functions and/or primary gaming states;

providing an Application Programming Interface enabling communication from a distal intelligence source to the gaming operating system;

communicating with the Application Programming Interface to the functions and/or primary gaming states in the library of the gaming operating system by providing a Makefile or other procedure for building a gaming application, and a configuration file for running the gaming operation system on a proximal computing system;

providing gaming specific data relating to at least one specific gaming application; and

compiling a program specific to at least one gaming application that is compatible with the gaming operating system.

2. The method of assisting in the development of a computer based wagering gaming application according to claim 1 comprising the steps of:

wherein the library of at least two software gaming elements comprises gaming elements selected from the group consisting of random number generator, game initiation sequence, bonus module, video gaming module, audio gaming module, jackpot module, graphics conversion tool, debugging tool, pay-out table module, value-handling module, power-loss recovery module, gaming payout history module, player history module, and user interaction module.

3. The method of claim 2 wherein public and/or private authentication keys are revved and different public and/or private authentication keys is provided to each of at least two different legal jurisdictions.

4. The method of claim 1 wherein the computerized wagering game application is developed for an apparatus comprising:

a computerized game controller operable to control the computerized wagering game having a processor, memory, and nonvolatile storage; and

an operating system comprising: a system handler application which provides gaming related functions and services to game programs; and

an operating system kernel that executes the system handler application.

5. The method of claim 4 wherein the system handler application comprises at least one system selected from the group consisting of

a) a plurality of device handlers,

b) software having the ability when executed to:

load a gaming program; and

execute the new gaming program;

c) an API with functions callable from the game program;

d) an event queue;

e) a game personality described in a selected mode; and

f) a combination of an event queue that determines the order of execution of each specified device handler; an API having a library of functions; an event queue capable of queuing on a first come, first serve basis; and an event queue capable of queuing using more than one criteria.

6. The method of claim 4 wherein game data modified by gaming program objects is stored in nonvolatile storage or wherein the system handler and kernel work in communication to hash system handler code and operating system kernel code.

7. The method of claim 6, wherein game data modified by gaming program objects is stored in nonvolatile storage and changing game data in nonvolatile storage causes execution of a corresponding callback function in the system handler application.

8. The method of claim 4, wherein the operating system kernel is a Linux operating system kernel having customized proprietary modules and the kernel has at least one modification wherein each modification is selected from the group consisting of: 1) accessing user level code from ROM, 2) executing from ROM, 3) zero out unused RAM, 4) test and/or hash the kernel, and 5) disabling selected device handlers.

9. The method of claim 4 wherein the apparatus contains a machine-readable element with machine-readable instructions thereon, the instructions when executed operable to cause the processor to manage at least one gaming program object via a system handler application and to execute a single gaming program object at any one time, wherein gaming program objects are operable to share game data in nonvolatile storage within the processor in the computerized wagering game system.

10. The method of claim 9 wherein programming directs the gaming apparatus to effect a procedure selected from the group consisting of a) only one gaming program object executes at any one time, b) there are instructions operable when executed to cause a computer to provide functions through an API that comprises a part of the system handler application, and c) when instructions are executed, the instructions are operable to store game data in nonvolatile storage, such that the state of the computerized wagering game system is maintained when the machine loses power.

11. A method wherein after compiling a program specific to at least one gaming application that is compatible with the gaming operating system according to claims 1, 2, 3, 4 or 5, the program specific to at least one gaming application of manages data in the computerized wagering game apparatus via a system handler application comprising:

loading a shared object,

executing the shared object, and

accessing and storing game data in nonvolatile storage.

12. The method of claim 11 wherein managing data further comprises a) unloading the first program object, and loading a second program object or b) executing a corresponding callback function upon alteration of game data in nonvolatile storage.

13. The method of claim 1 wherein after compiling a program specific to at least one gaming application that is compatible with the gaming operating system, a machine-readable memory storage element with instructions thereon has the instructions executed to cause a computer to:

load a first program shared object,

execute a first program shared object,

store gaming data in nonvolatile storage, such that a second program object later loaded can access gaming data variables in nonvolatile storage,

unload the first program shared object from system memory, and

load the second program shared object to system memory so that the second program shared object is accessible to the computer as instructions.

14. The method of claim 13 wherein additional instructions are executed to cause a computer to perform a task selected from the group consisting of a) executing a corresponding callback function upon alteration of game data in the nonvolatile storage; and b) managing events via the system handler application.

15. The method of claim 1 wherein the computerized wagering game application is developed for an apparatus comprising:

a universal operating system stored in a memory storage component that may be operatively inserted along with game identity data into an electronic or electromechanical gaming device having ancillary functions so that the gaming device can effect play of the game provided in the game identity data and the operating system will control at least one ancillary function selected from the group consisting of coin acceptance, credit acceptance, currency acceptance and boot up, the gaming device having at least one system handler application, and the operating system comprising a system handler and an operating system kernel.

16. The method of claim 15 wherein the apparatus further comprises at least one of a plurality of APIs, an operating system kernel customized for gaming purposes, and an event queue.

17. The method of claim 15 wherein the apparatus further comprises a system handler comprising a plurality of device handlers or the operating system controls a networked on-line system or control a progressive meter.

18. The method of claim 15 wherein the operating system comprises a kernel customized for gaming purposes utilizing a method of operation selected from the group consisting of: 1) accessing user level code from ROM, 2) executing from ROM, 3) zero out unused RAM, 4) test and/or hash the kernel, and 5) disabling selected device handlers.

19. A method comprising a) customizing an operating system kernel and b) providing the customized kernel of the operating system into a gaming apparatus, at least one customization being effected to obtain functionality of the gaming apparatus, the customization being a kernel modification for a process selected from the group consisting of:

1) accessing user level code from ROM;
2) executing user level code from ROM;
3) zeroing out unused RAM;
4) testing and/or hashing the kernel; and
5) disabling selected device handlers.

20. A method of converting a first game that operates on a first gaming system so that the game operates on a universal gaming system, the method comprising:

removing a first game operating system from the gaming system, the first game operating system including hardware and software;

installing the universal gaming system of claim 15 in place of the game operating system, the universal gaming system including a game program layer, an open operating system, and a game controller for running the game program layer on the open operating system;

providing functional interfaces between the universal gaming system and game devices; and

installing a second game specific program in the game program layer configured to operate with the open operating system.

21. The method of claim 20, comprising at least one step selected from the group consisting of:

a) providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the game devices via the system application handler;

b) configuring the system handler application to include one or more device handlers for interfacing with the game devices, wherein at least one of the device handlers operates as a protocol manager between the games device and the open operating system;

c) providing the open operating system to include an operating system kernel that executes the system handler application; and

d) providing the game program layer with at least one gaming program object.

22. The method of claim 21, wherein the at least one gaming program object is specific to a type of game played on the universal gaming system.

23. The method of claim 20, comprising at least one step selected from the group consisting of :

changing a type of game played on the universal gaming system by changing game program objects;

configuring the game program layer to operate the game as a slot machine;

operating the slot machine as a mechanical reel-based slot machine; and

configuring the open operating system to include a resource manager for mapping game specific resources.

24. The method of claim 23 including mapping game specific resources by parsing a configuration file, mapping operating system resources based on the configuration file, and storing the resource map in memory.

25. The method of claim 24, wherein mapping operating system resources based on the configuration file includes mapping input/output lines to system resources.

26. The method of claim 22 comprising converting the first gaming system from a cash accepting gaming system to a cashless gaming system, the method including providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the game devices accomplished via the system application handler, and configuring the system handler application to include one or more device handlers for interfacing with the game devices, the configuring including installing a card reader device handler, and installing a card reader in communication with the card reader device handler, and optionally including configuring the system handler application to include a ticket printer device handler; and installing a ticket printer in communication with the ticket printer device handler.

27. The method of claim 22 wherein a slot machine game operating system is removed from the first gaming system and the functional interfaces are between the universal gaming system and slot machine game devices.

28. The method of claim 27 comprising at least one step selected from the group consisting of:

a) providing the open operating system with a system application handler, wherein the functional interfaces include a functional interface between the gaming system and the slot machine game devices via the system application handler;

b) configuring the system handler application to include one or more device handlers for interfacing with the slot machine game devices, wherein at least one of the device handlers operates as a protocol manager between the slot machine games device and the open operating system;

c) configuring an I/O device handler to interface with slot machine input devices and slot machine output devices;

d) providing slot machine input devices that include a mechanical arm, button acceptor and coin acceptor;

e) providing the slot machine with output devices inclusive of slot machine reels, credit displays, and speakers.

29. The method of claim 28, comprising converting the mechanical reel slot machine game having only cash, token, credit balance and currency acceptance capability to a cashless gaming system via the system handler application, the converting including providing a card reader device handler, and installing a card reader in communication with the card reader device handler and optionally providing a ticket printer device handler, and installing a ticket printer in communication with the ticket printer device handler.

30. A method of configuring a game program layer for a universal gaming system that is configured for a game program layer and an open operating system, the method comprising:

configuring the game program layer on a computer remote from a first non-universal gaming system; and

downloading the game program layer into the universal gaming system and performing at least one sequence comprising

a) defining a game template; and configuring the game program layer using the game template;

b) storing the game program on a removable media card;

c) providing removable media as flash memory.

31. The method of claim 30 wherein the game program is stored on a removable media card and the removable media card is plugged into the gaming system, and then running the game program layer via the open operating system from the removable media card.

32. The method of claim 30 comprising an additional step of preparing the game program layer for authentication by plugging the removable media card into an authenticating system.

33. The method of claim 30 performed as a network based method of providing a game program layer for a universal gaming system configured for remote operation using an open operating system, the method including defining a user interface to communicate between the remote computer and the universal operating system.

34. The method of claim 33 wherein the game program layer is configured to use user interface remote from the gaming system or via a web page template at the user interface.

35. A gaming system developed for use in a casino by the method of claim 1 comprising:

a game controller configured to operate the gaming system; and

a first nonvolatile memory and a second nonvolatile memory for storing critical gaming information, wherein the first nonvolatile memory and the second nonvolatile memory are configured to communicate with the game controller as a gaming RAID system for redundant storage of critical gaming information. RAID not defined in text,

the gaming system enabling redundant NVRAM storage to be replaceable while operating power for the system is on.

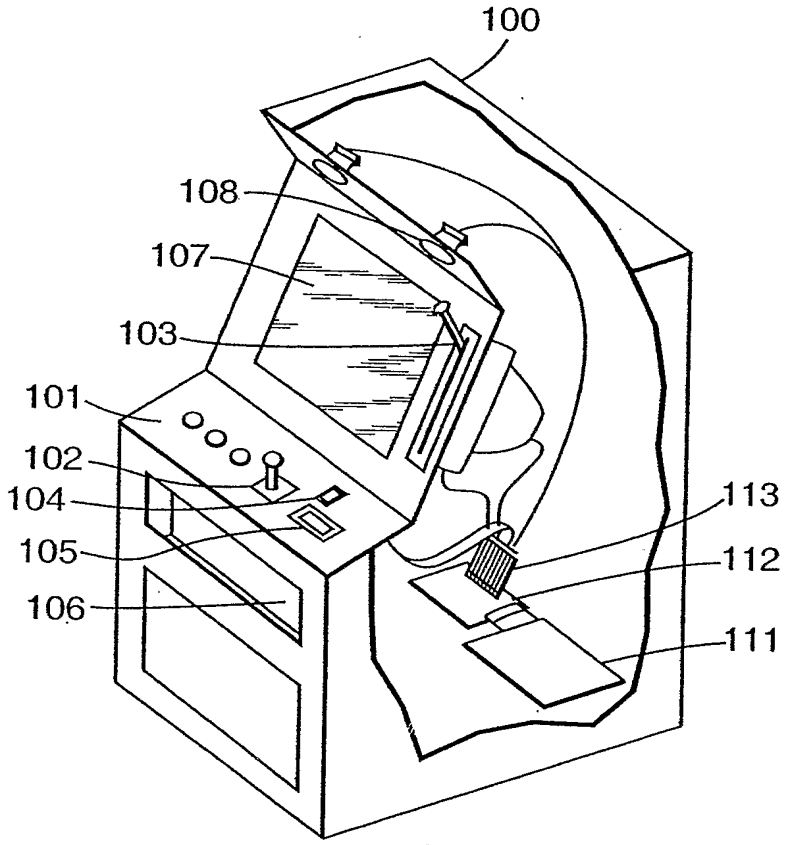


Fig. 1

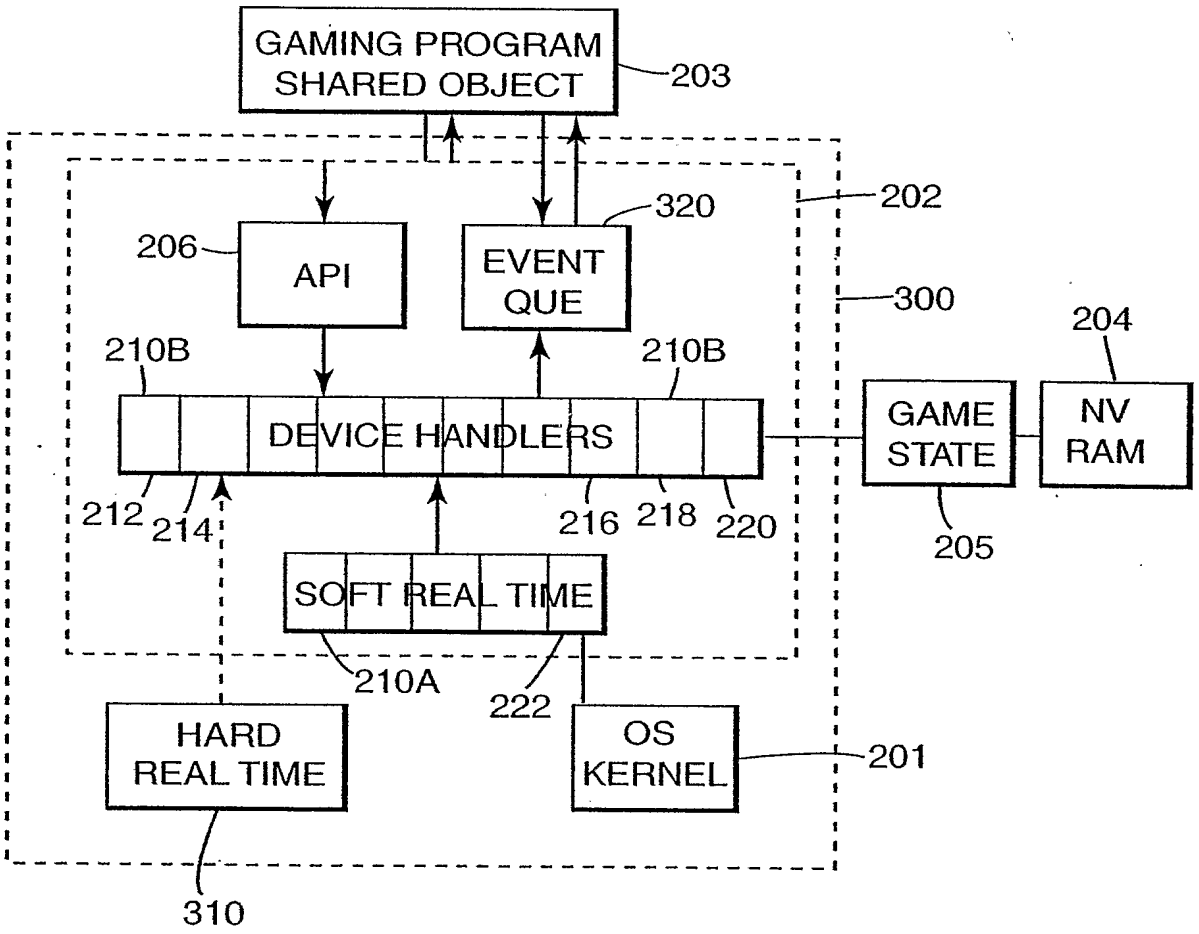

GAMING PROGRAM SHARED OBJECT
203
202
320
206
API
EVENT QUE
300
204
210B
210B
DEVICE HANDLERS
GAME STATE
NV RAM
212
214
216
218
220
205
SOFT REAL TIME
210A
222
HARD REAL TIME
OS KERNEL
201
310

Fig. 2

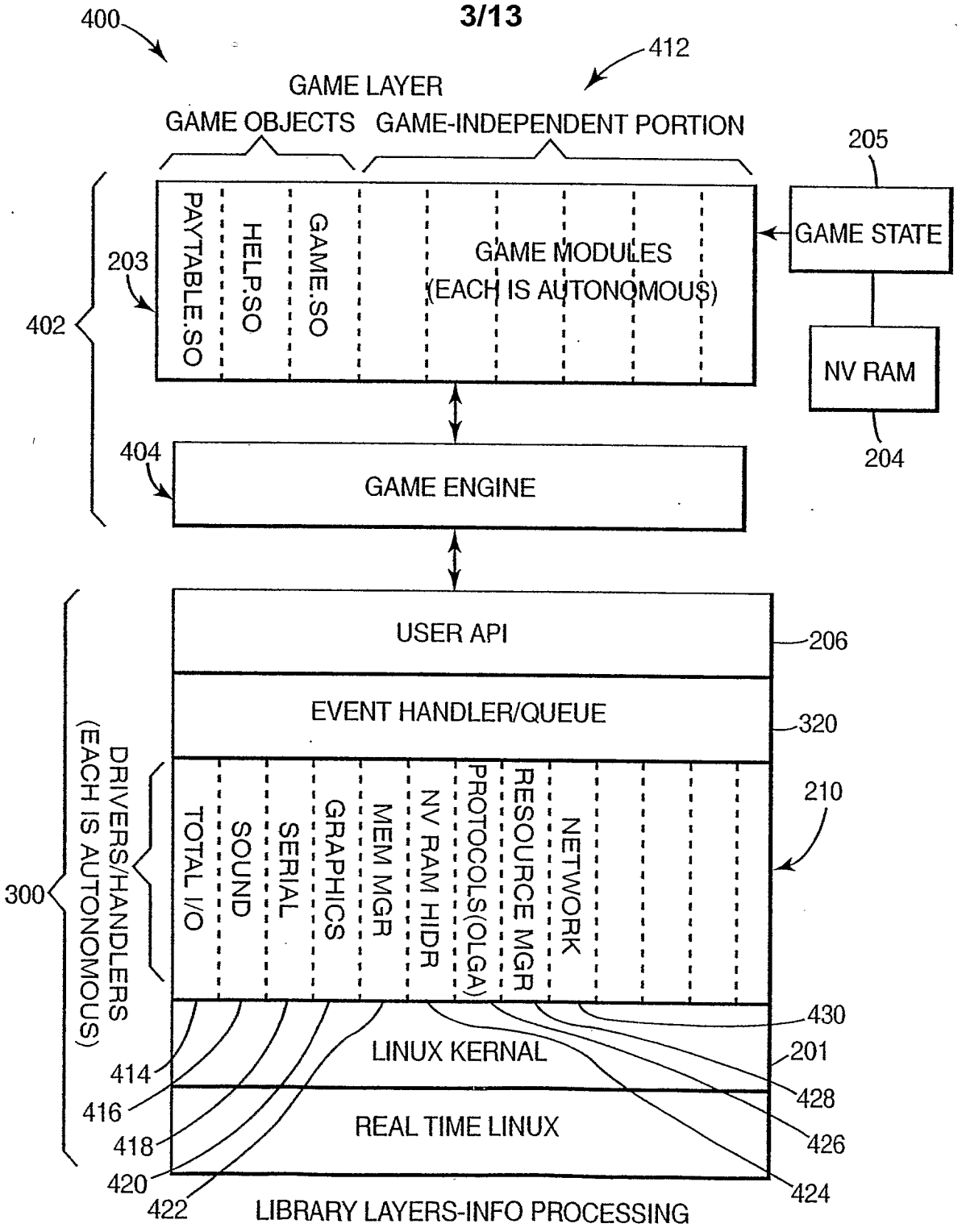

400
412
GAME LAYER
GAME OBJECTS
GAME-INDEPENDENT PORTION
205
PAYTABLE.SO
HELP.SO
GAME.SO
GAME MODULES
(EACH IS AUTONOMOUS)
GAME STATE
203
402
NV RAM
204
404
GAME ENGINE
USER API
206
EVENT HANDLER/QUEUE
320
DRIVERS/HANDLERS
(EACH IS AUTONOMOUS)
300
TOTAL I/O
SOUND
SERIAL
GRAPHICS
MEM MGR
NV RAM HIDR
PROTOCOLS(OLGA)
RESOURCE MGR
NETWORK
210
430
LINUX KERNAL
201
414
416
418
420
422
428
426
424
REAL TIME LINUX
LIBRARY LAYERS-INFO PROCESSING

Fig. 3

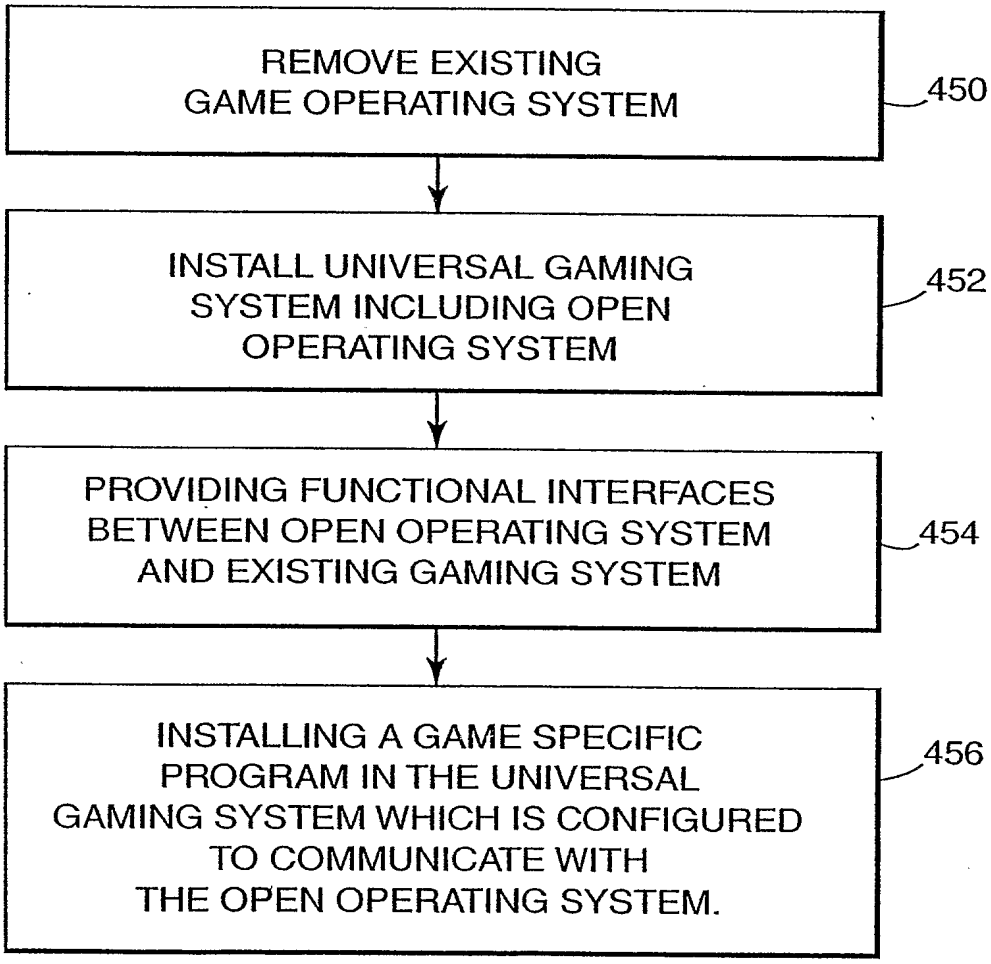

REMOVE EXISTING GAME OPERATING SYSTEM
450
INSTALL UNIVERSAL GAMING SYSTEM INCLUDING OPEN OPERATING SYSTEM
452
PROVIDING FUNCTIONAL INTERFACES BETWEEN OPEN OPERATING SYSTEM AND EXISTING GAMING SYSTEM
454
INSTALLING A GAME SPECIFIC PROGRAM IN THE UNIVERSAL GAMING SYSTEM WHICH IS CONFIGURED TO COMMUNICATE WITH THE OPEN OPERATING SYSTEM.
456

Fig. 4

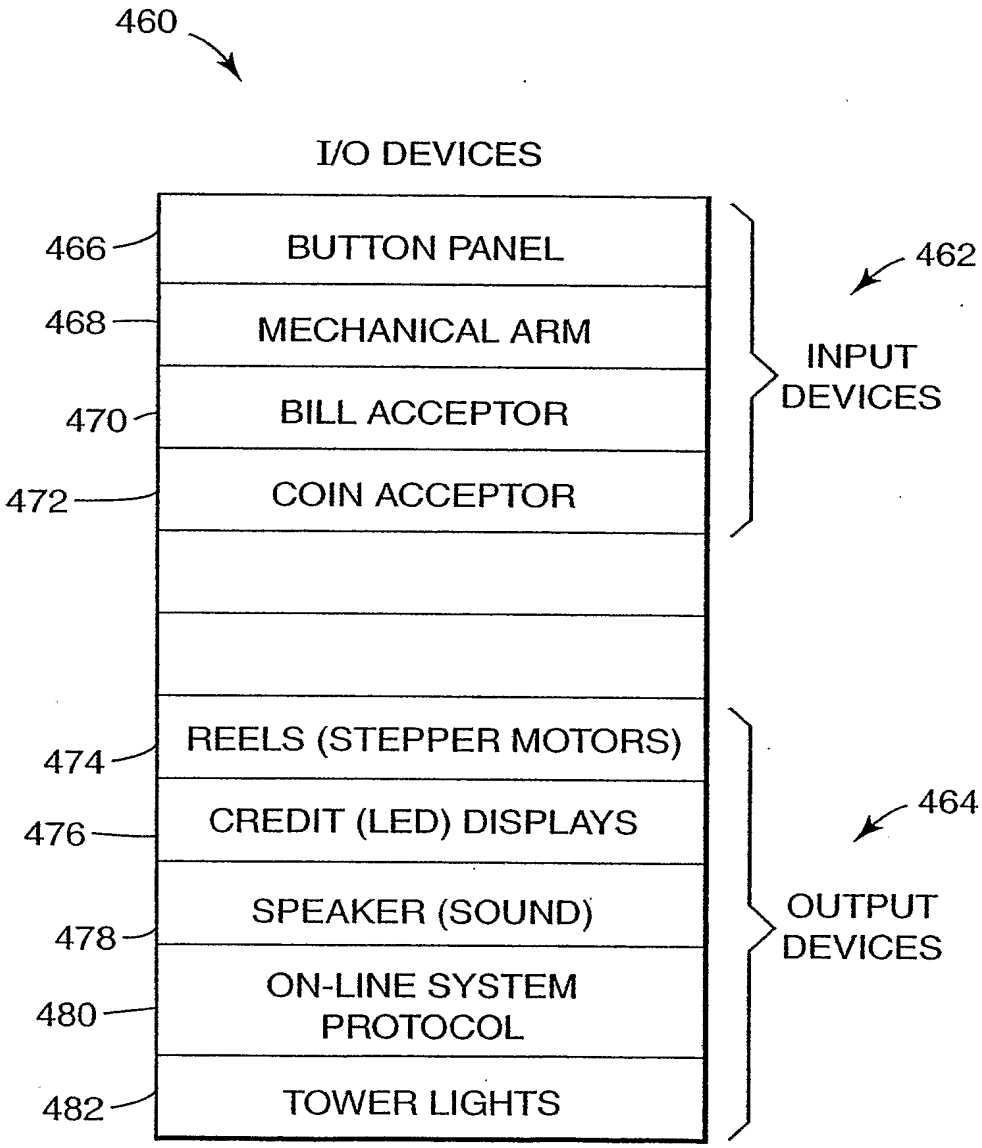

460
I/O DEVICES
466
BUTTON PANEL
468
MECHANICAL ARM
470
BILL ACCEPTOR
472
COIN ACCEPTOR
462
INPUT DEVICES
474
REELS (STEPPER MOTORS)
476
CREDIT (LED) DISPLAYS
478
SPEAKER (SOUND)
480
ON-LINE SYSTEM PROTOCOL
482
TOWER LIGHTS
464
OUTPUT DEVICES

Fig. 5

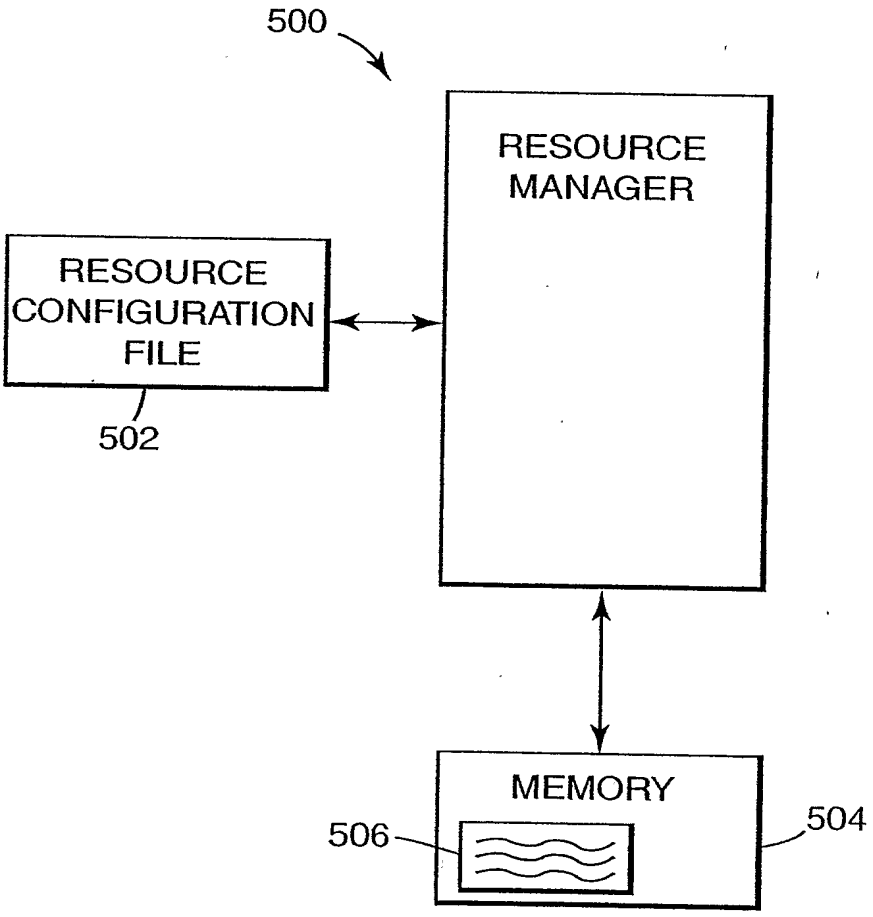

500
RESOURCE MANAGER
RESOURCE CONFIGURATION FILE
502
MEMORY
506
504

Fig. 6

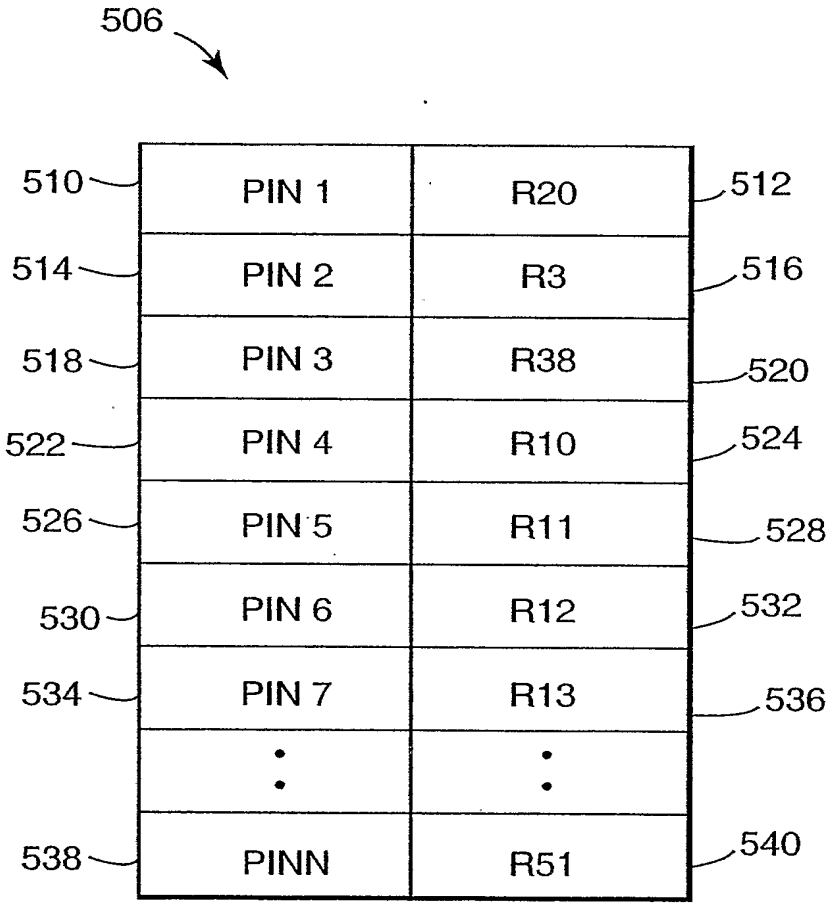

506
510
PIN 1
R20
512
514
PIN 2
R3
516
518
PIN 3
R38
520
522
PIN 4
R10
524
526
PIN 5
R11
528
530
PIN 6
R12
532
534
PIN 7
R13
536
538
PINN
R51
540

Fig. 7

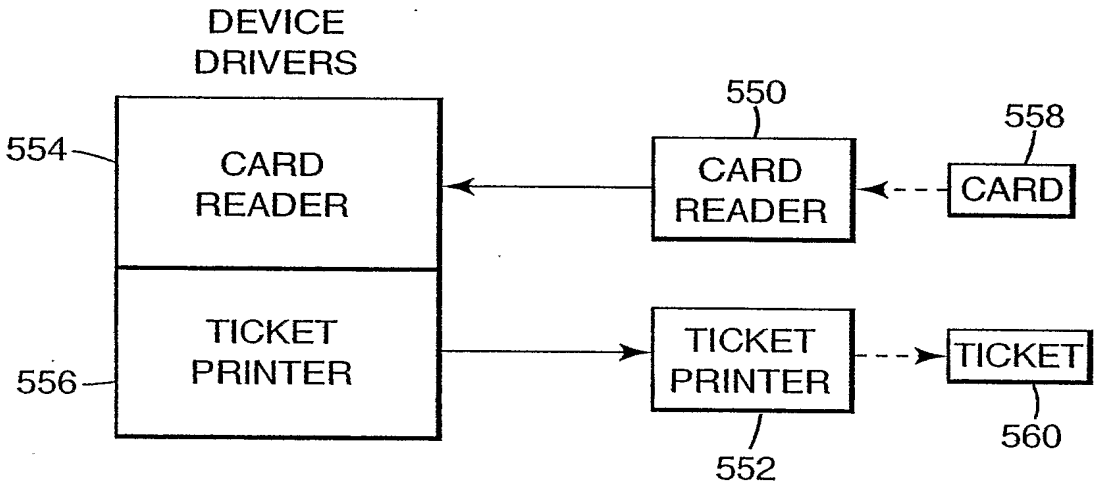

DEVICE
DRIVERS
554
CARD
READER
556
TICKET
PRINTER
550
CARD
READER
558
CARD
TICKET
PRINTER
552
TICKET
560

Fig. 8

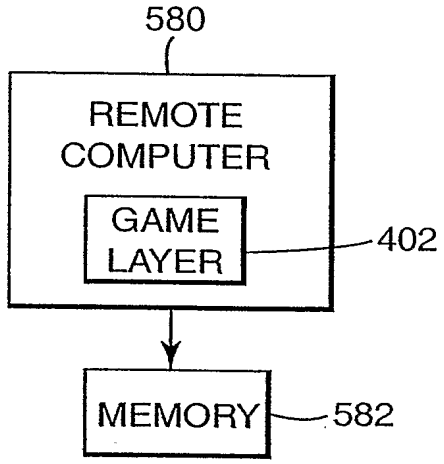

580
REMOTE COMPUTER
GAME LAYER
402
MEMORY
582

Fig. 9

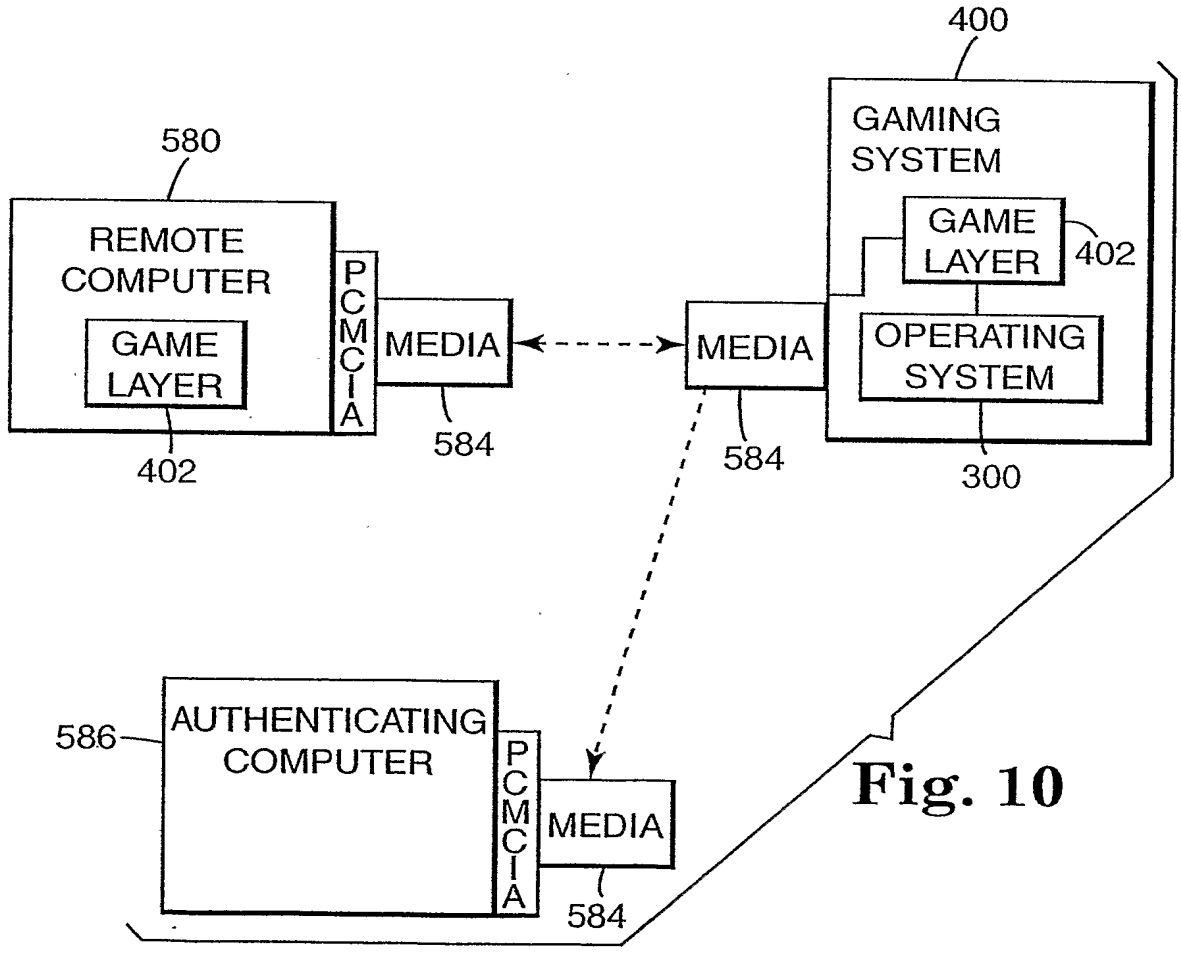

400
GAMING SYSTEM
GAME LAYER
402
OPERATING SYSTEM
300
580
REMOTE COMPUTER
GAME LAYER
402
PCMCIA
MEDIA
584
MEDIA
584
586
AUTHENTICATING COMPUTER
PCMCIA
MEDIA
584

Fig. 10

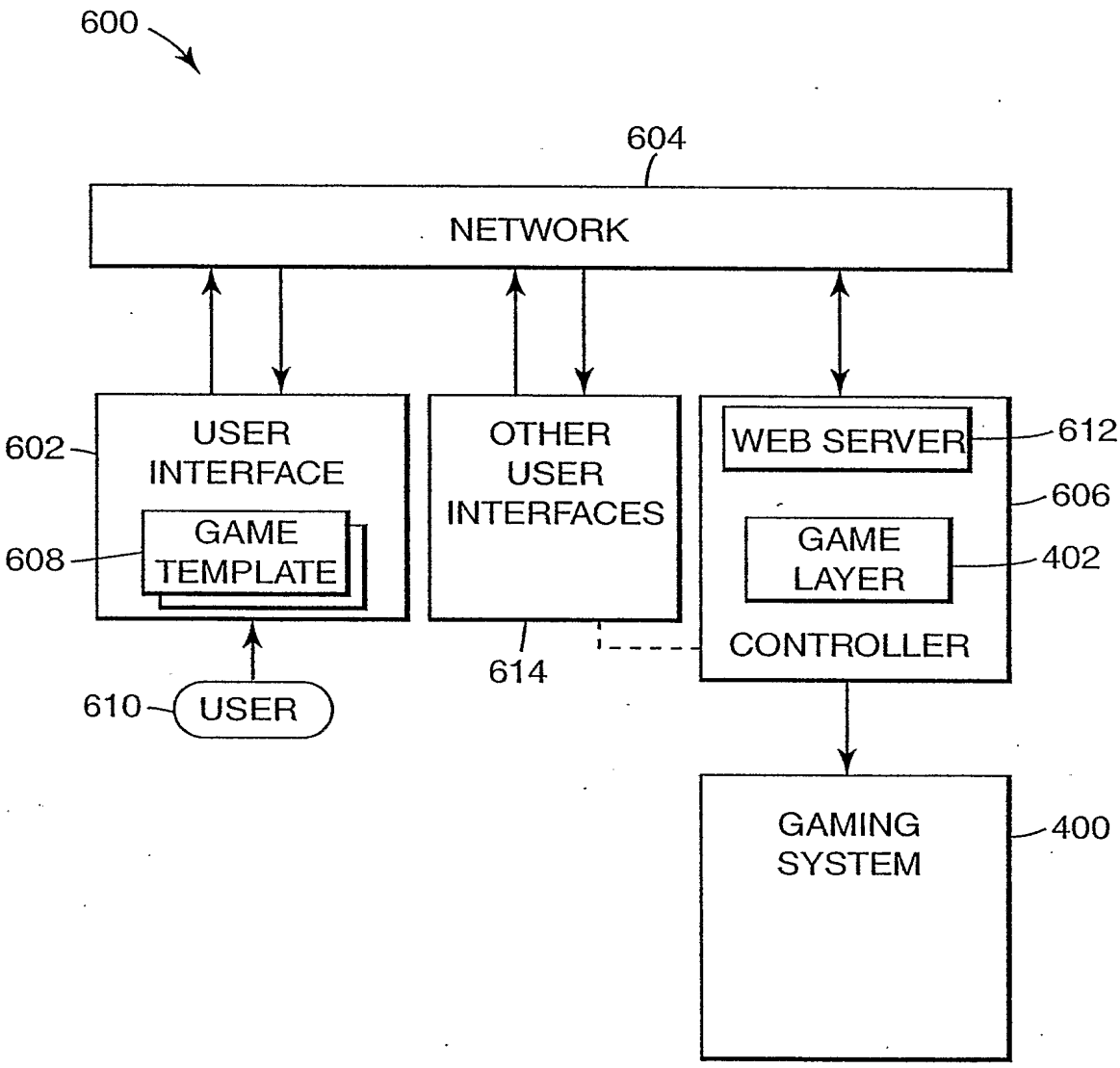

600
604
NETWORK
602
USER INTERFACE
608
GAME TEMPLATE
OTHER USER INTERFACES
614
WEB SERVER
612
606
GAME LAYER
402
CONTROLLER
610
USER
GAMING SYSTEM
400

Fig. 11

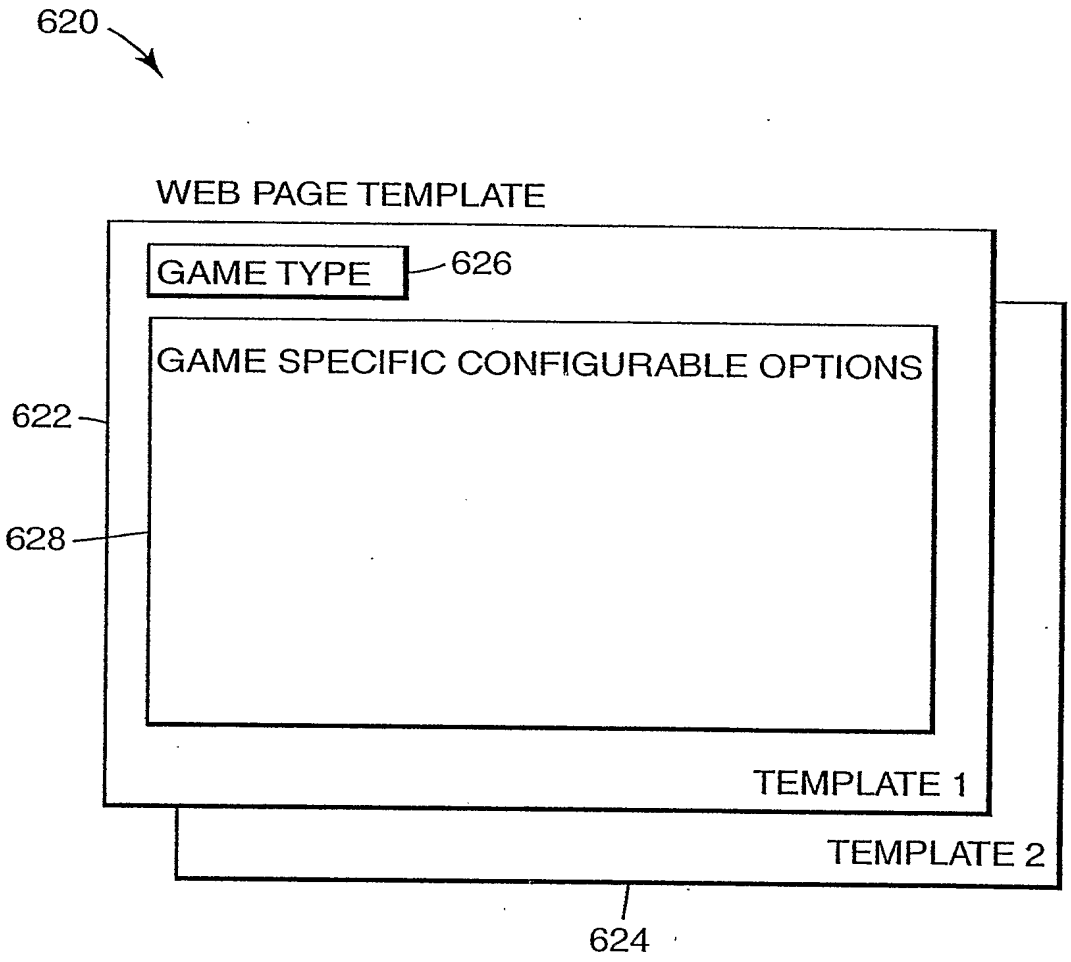

620
WEB PAGE TEMPLATE
GAME TYPE
626
GAME SPECIFIC CONFIGURABLE OPTIONS
622
628
TEMPLATE 1
TEMPLATE 2
624

Fig. 12

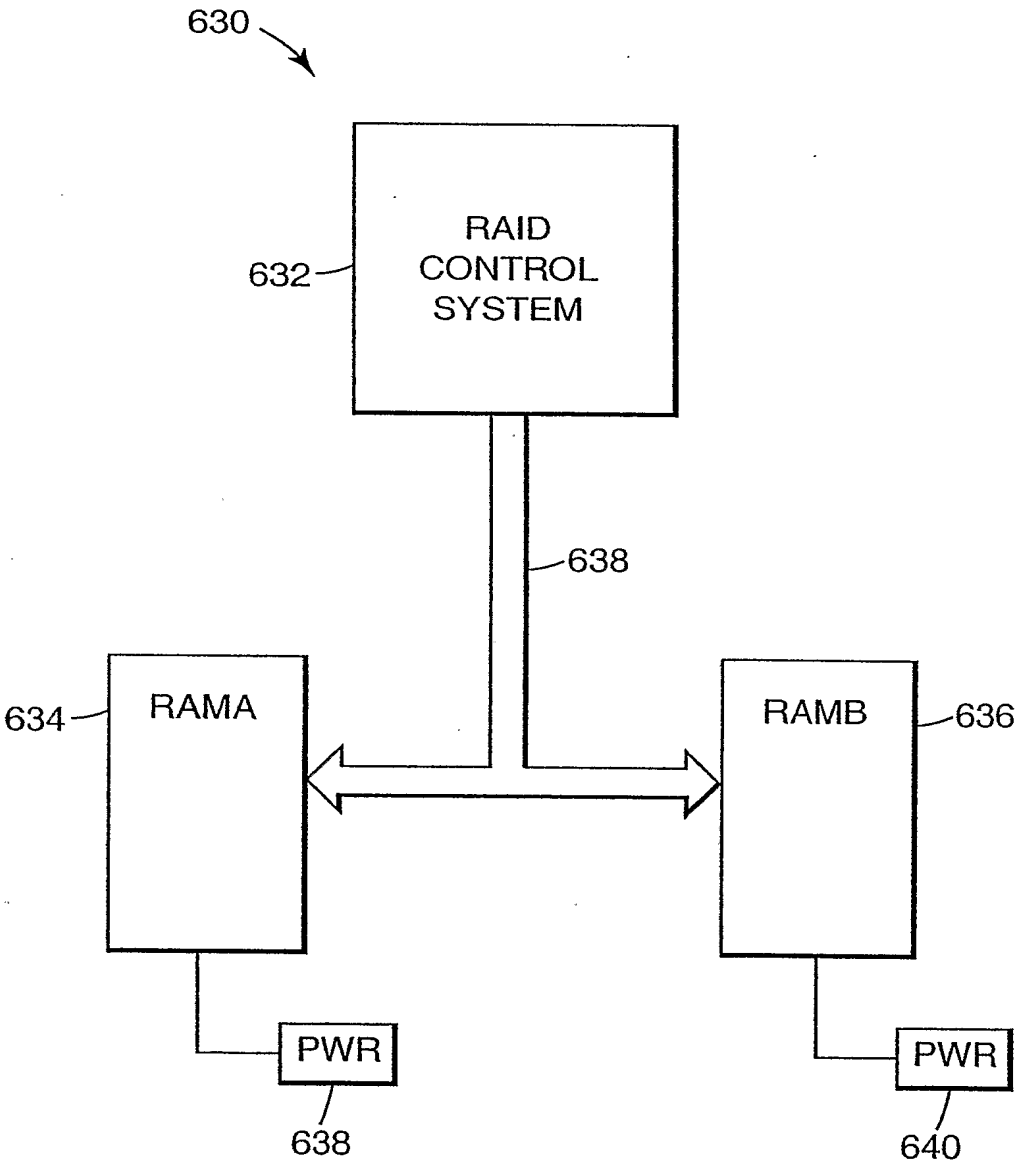
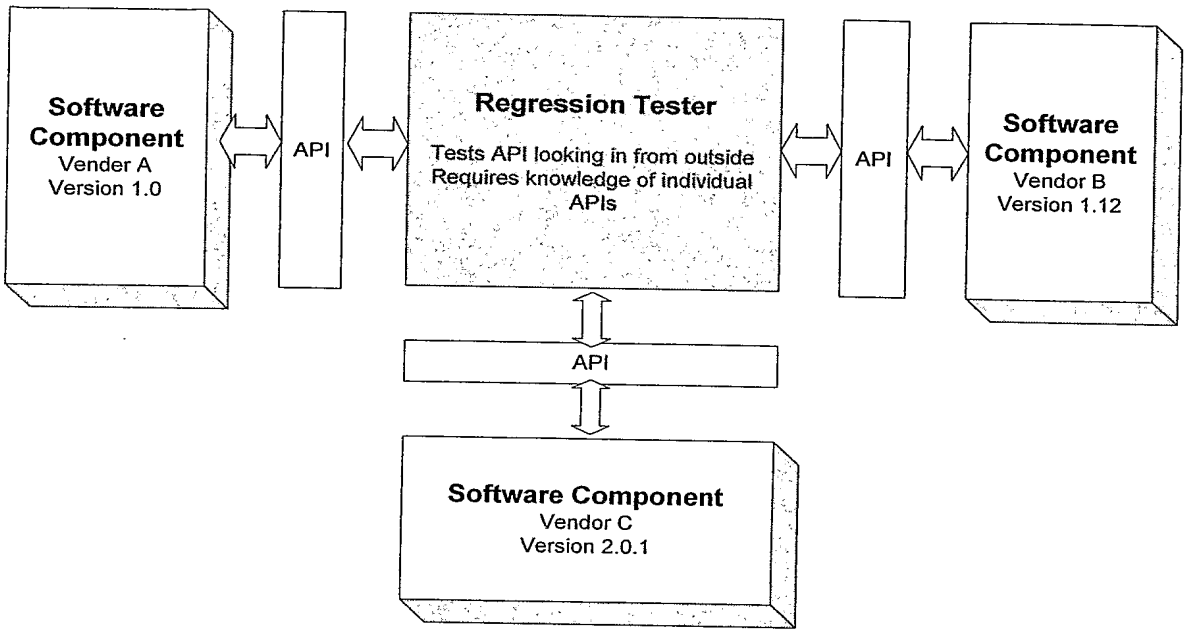

630
RAID
CONTROL
SYSTEM
632
638
RAMA
634
RAMB
636
PWR
638
PWR
640

Fig. 13


Software Component
Vender A
Version 1.0
API
Regression Tester
Tests API looking in from outside
Requires knowledge of individual APIs
API
Software Component
Vendor B
Version 1.12
API
Software Component
Vendor C
Version 2.0.1

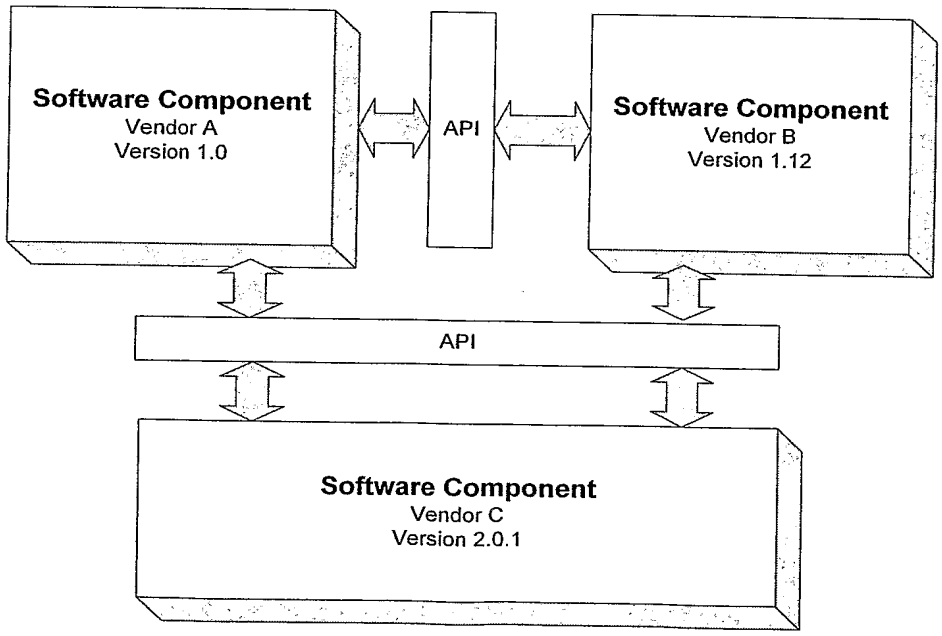

Software Component
Vendor A
Version 1.0
API
Software Component
Vendor B
Version 1.12
API
Software Component
Vendor C
Version 2.0.1

Fig. 14

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US02/30286

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) : G06F 17/00

US CL : 463/42

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 463/42, 43, 44, 47, 30, 31, 35, 37, 38, 22, 24, 25, 26, 27, 28, and 29.

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

Please See Continuation Sheet

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A, P	US 6,327,605 B2 (ARAKAWA et al) 04 December 2001(04.12.2001), see entire document.	1-35
A,P	US 6,319,125 B1 (ACRES) 20 November 2001 (10.11.2001), see entire document.	1-35

☐ Further documents are listed in the continuation of Box C.

☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

20 January 2003 (20.01.2003)

Date of mailing of the international search report

06 FEB 2003

Name and mailing address of the ISA/US

Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703)305-3230

Authorized officer

Kim Nguyen

Telephone No. (703)308-1078

Sheila Veney
Paralegal Specialist
Group 3700

Form PCT/ISA/210 (second sheet) (July 1998)

INTERNATIONAL SEARCH REPORT

PCT/US02/30286

Continuation of B. FIELDS SEARCHED Item 3:
EAST
search terms: gaming applications, wagering, interfaces, gaming operating system.